Red-Black Trees

Red-Black Trees…

… are Binary Search Trees that stay balanced (self balancing).

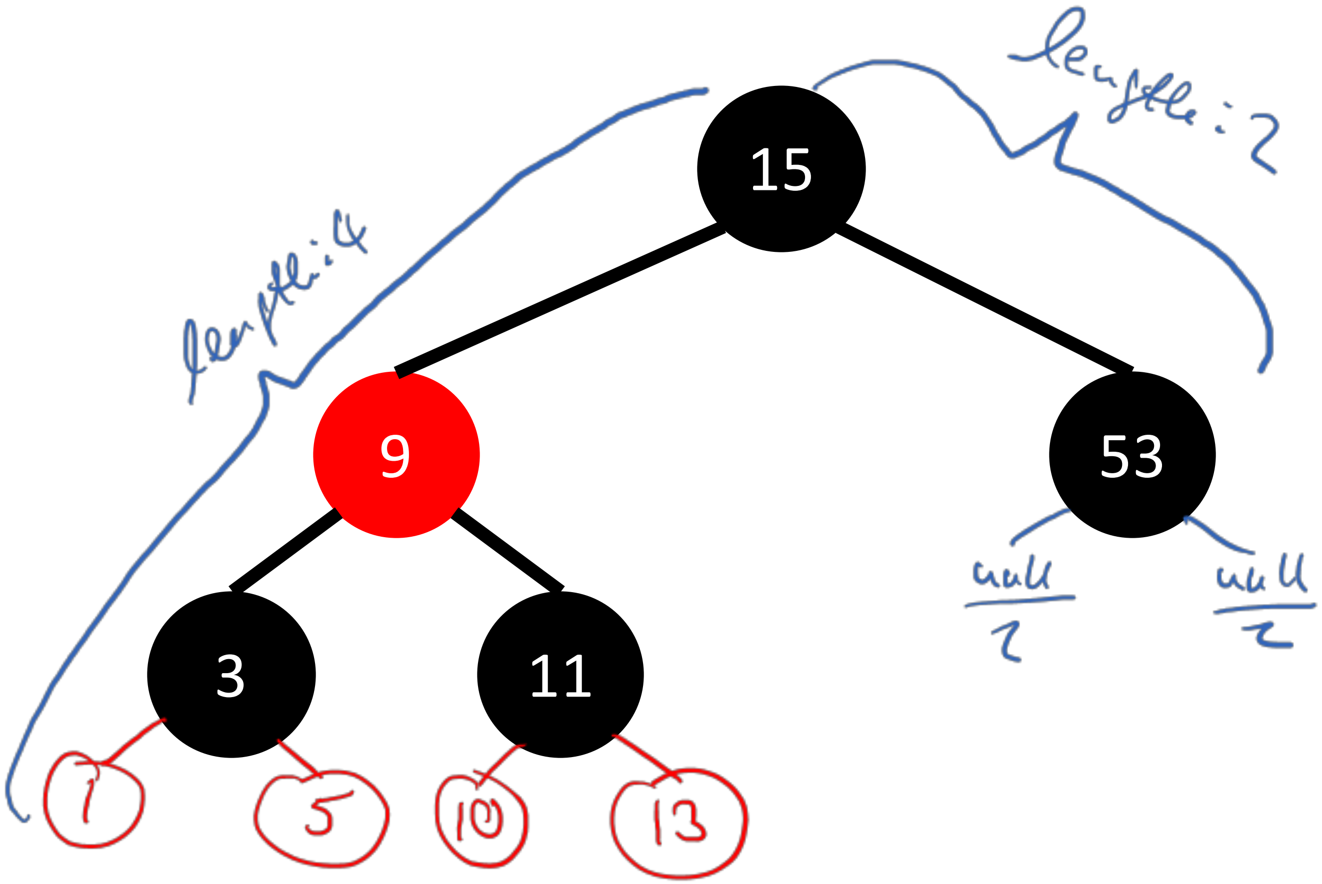
Red-Black Trees

Valid Red-Black Trees are regular BSTs with following properties:

- Each node is either **red** or **black**.
- The root node is **black**.
- No **red** nodes have **red** children.
- Every path from root to a null child has the same number of **black** nodes (black height of the tree).

Additional rule for Red-Black Trees:

- Null children are **black**.



(longest path root → null) ≤ 2 · (shortest path root → null)

Inserting into a Red-Black Tree

1. Insert new value using BST insertion algorithm
2. Color the new node **red**
3. Check Red-Black tree properties and restore if necessary

Which properties can be violated after insert?

- Each node is either **red** or **black**.
- The root node is **black**. !
- No **red** nodes can have **red** children. !
- Every path from root to ~~a to~~ a null child has the same number of **black** nodes (black height of the tree).

Repairing a Red Root

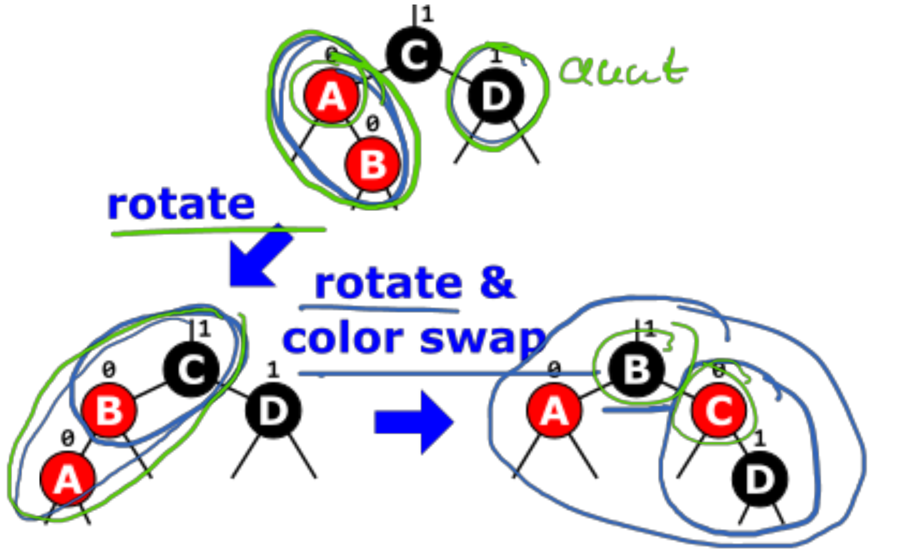
If the root of the tree is **red**, we can switch it to black without violating any other property.

Repairing Red Node With Red Child

We pick a repair operation:

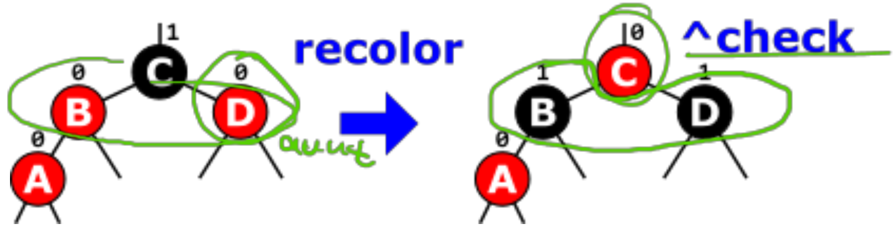
If aunt is **black** (or null)

→ rotate and color swap



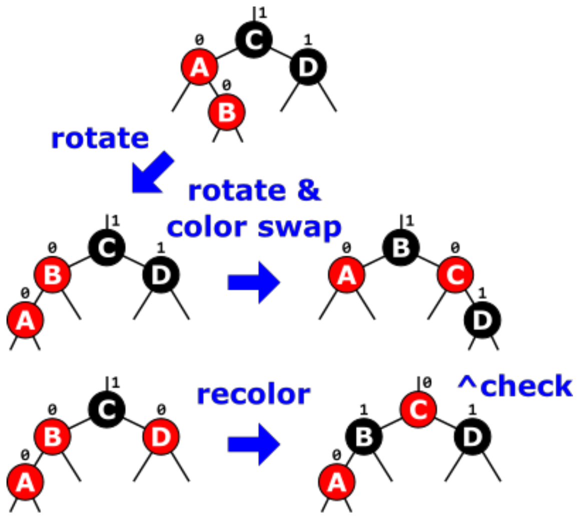
If aunt is **red**

→ recolor



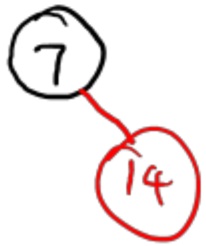
Red-Black Trees Insertion Example

Red-Black Tree Insertion Example (1)

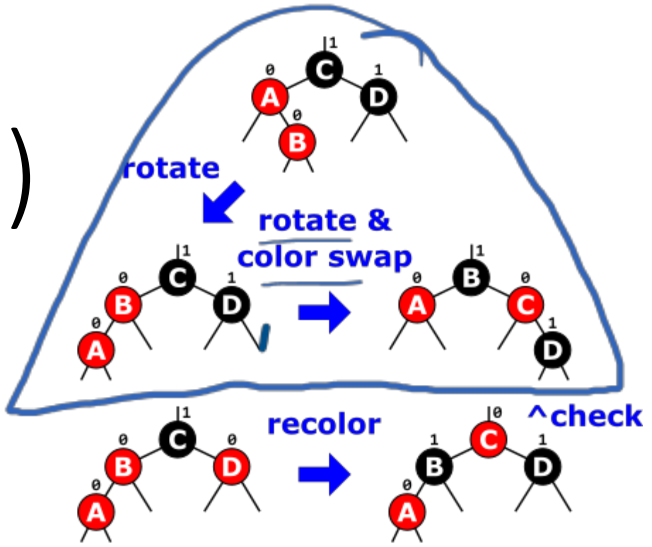


Insert: 7 and 14 into an empty tree

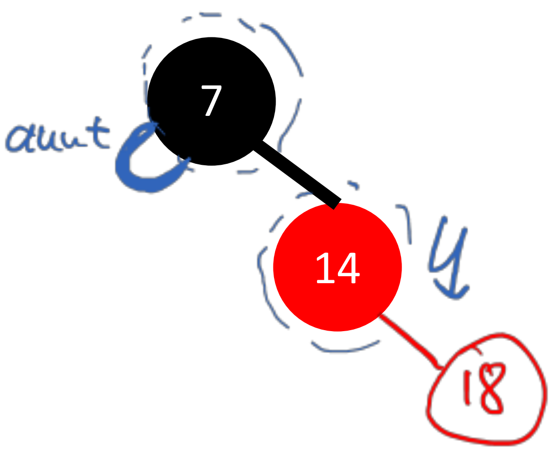
∅ $\xrightarrow{\text{insert 7}}$ 7 $\xRightarrow[\text{recolor 7 to black}]{\text{repair:}}$

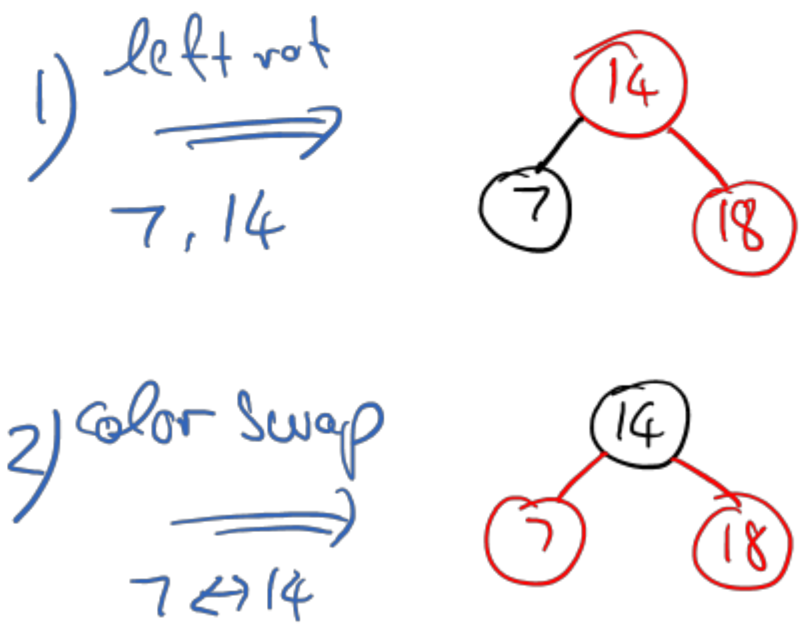


Red-Black Tree Insertion Example (2)



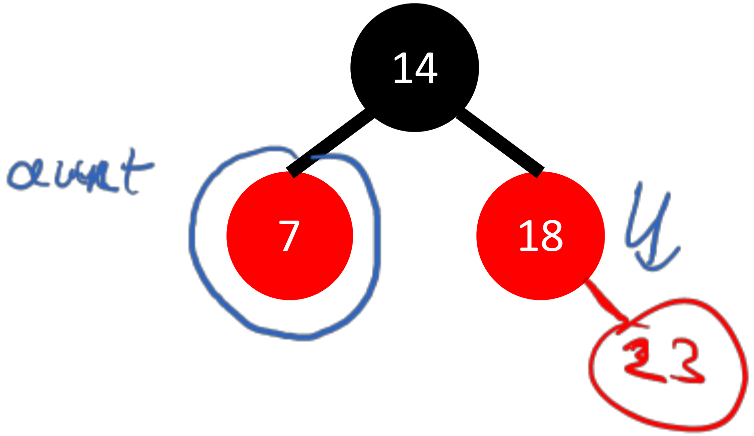
Insert: 18



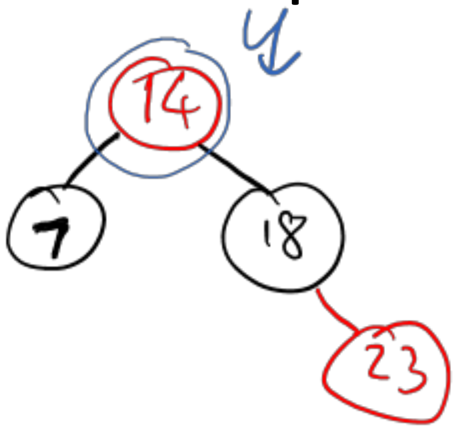


Red-Black Tree Insertion Example (3)

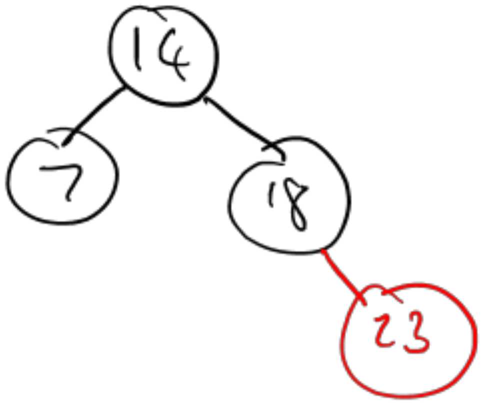
Insert: 23

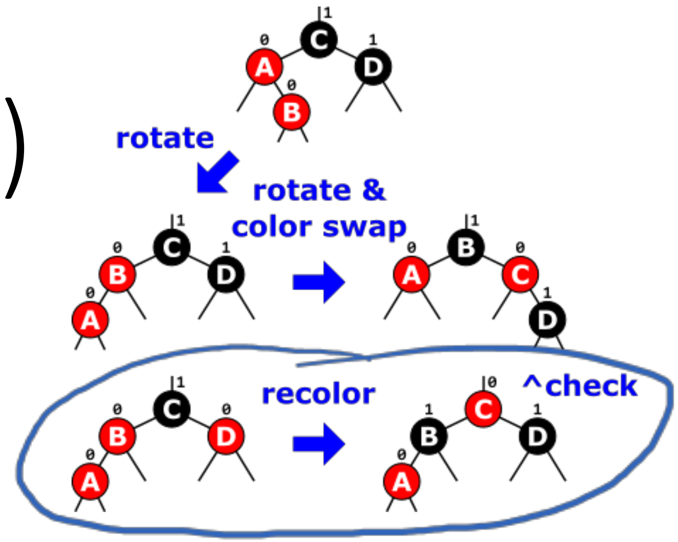


recolor ⟹



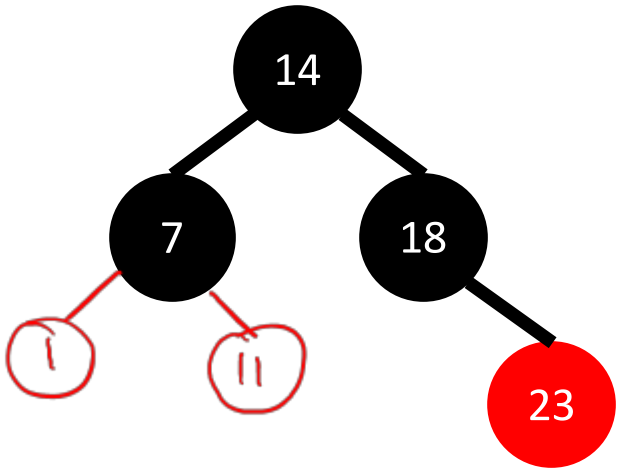
repair 2: ⟹ recolor 14 to check

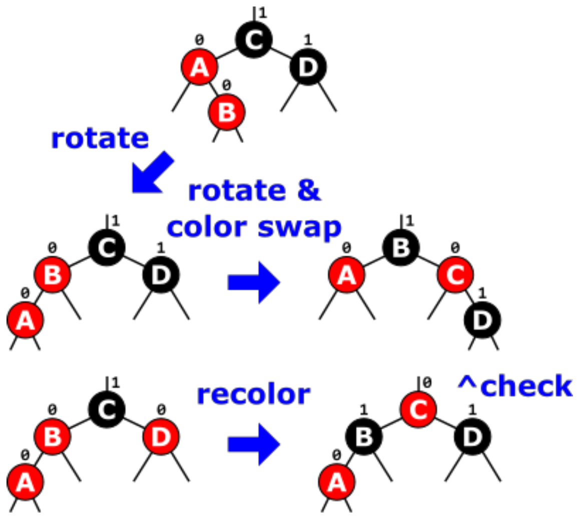




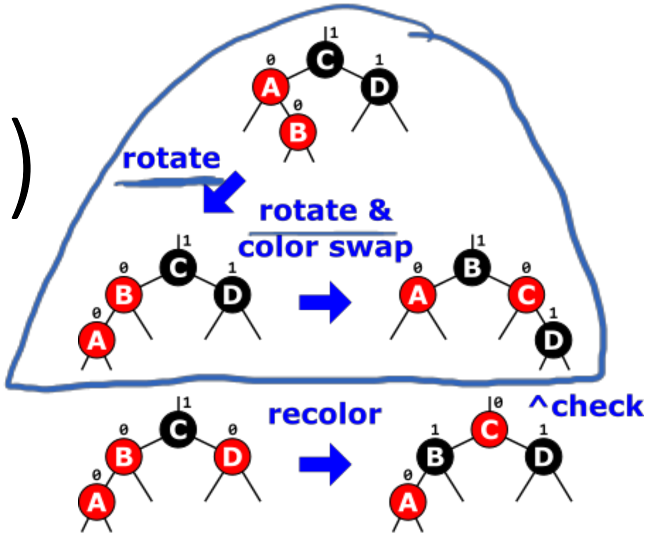
Red-Black Tree Insertion Example (4)

Insert: 1 and 11

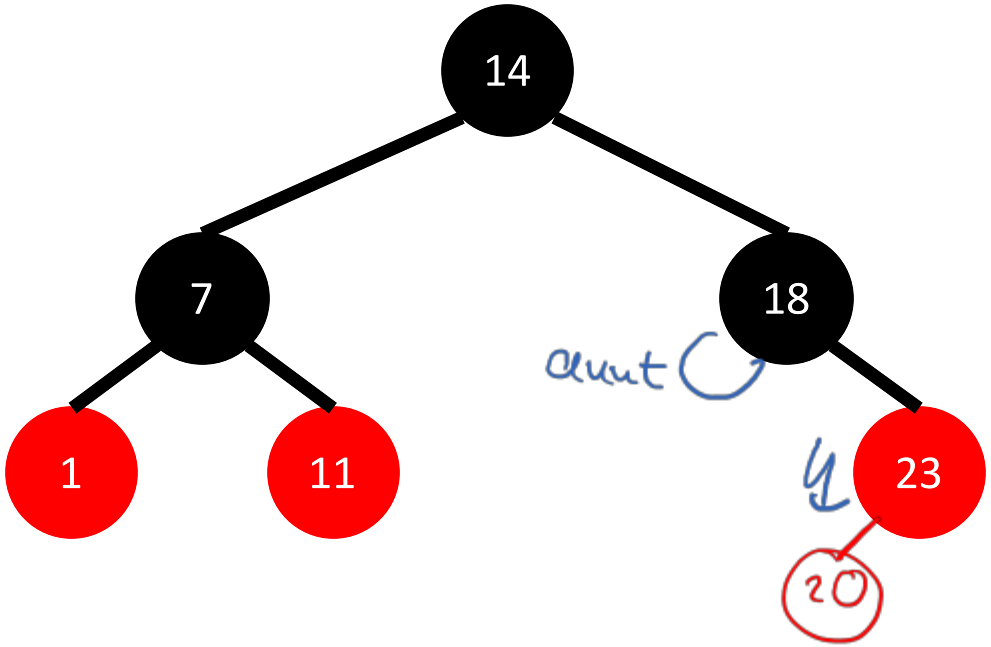




Red-Black Tree Insertion Example (5)

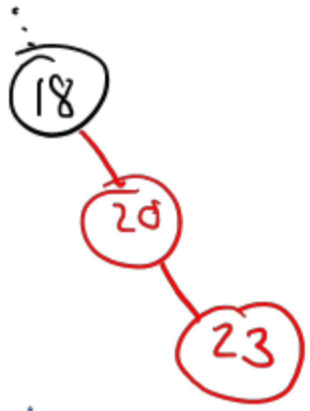


Insert: 20



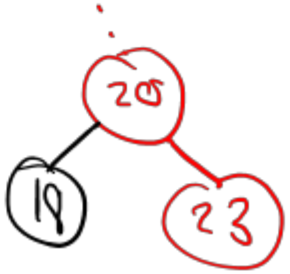
1) right rot

23, 20



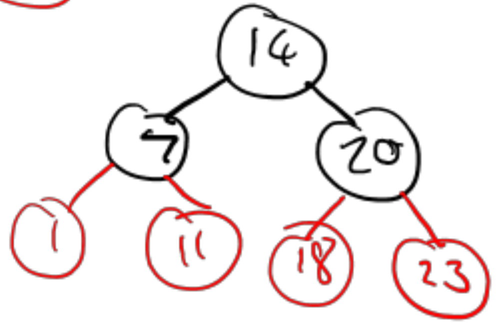
2) left rot

18, 20

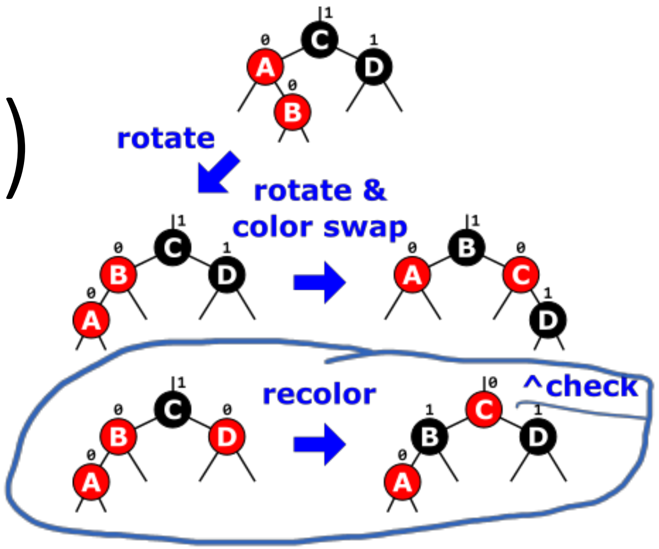


3) Color swap

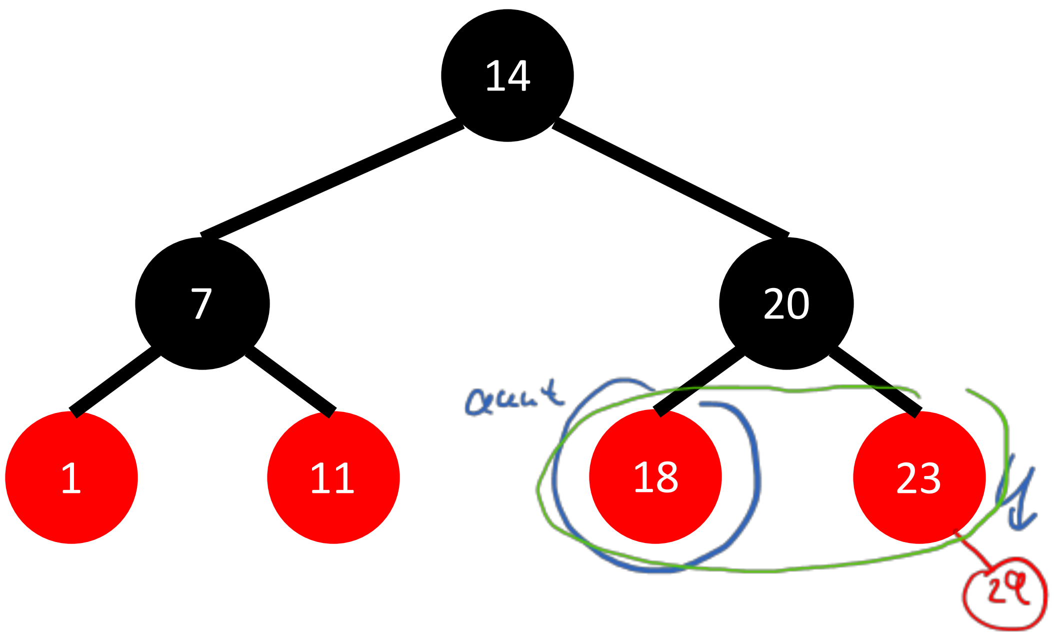
18, 20



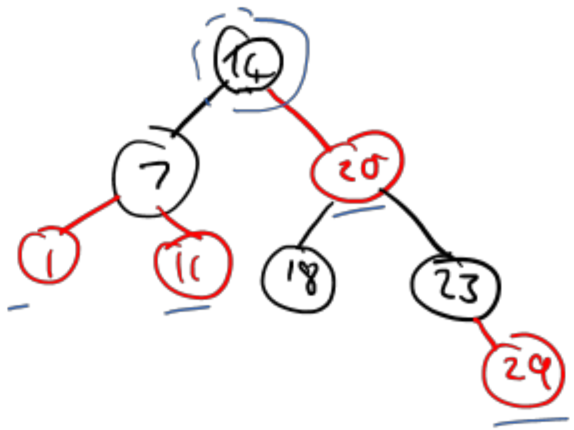
Red-Black Tree Insertion Example (6)



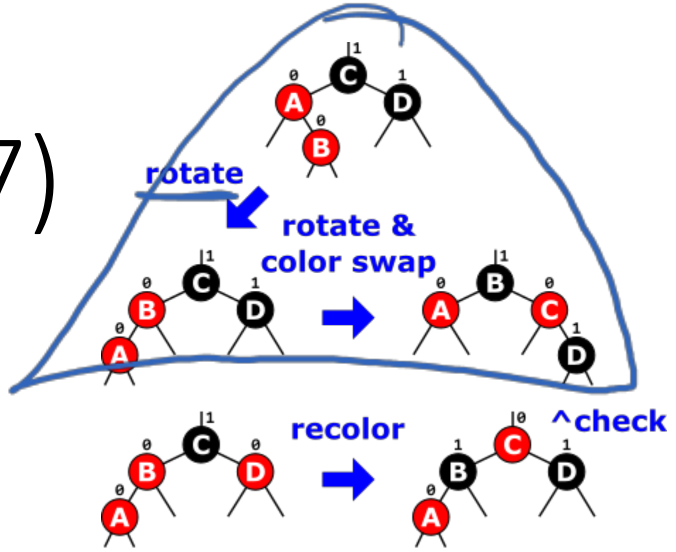
Insert: 29



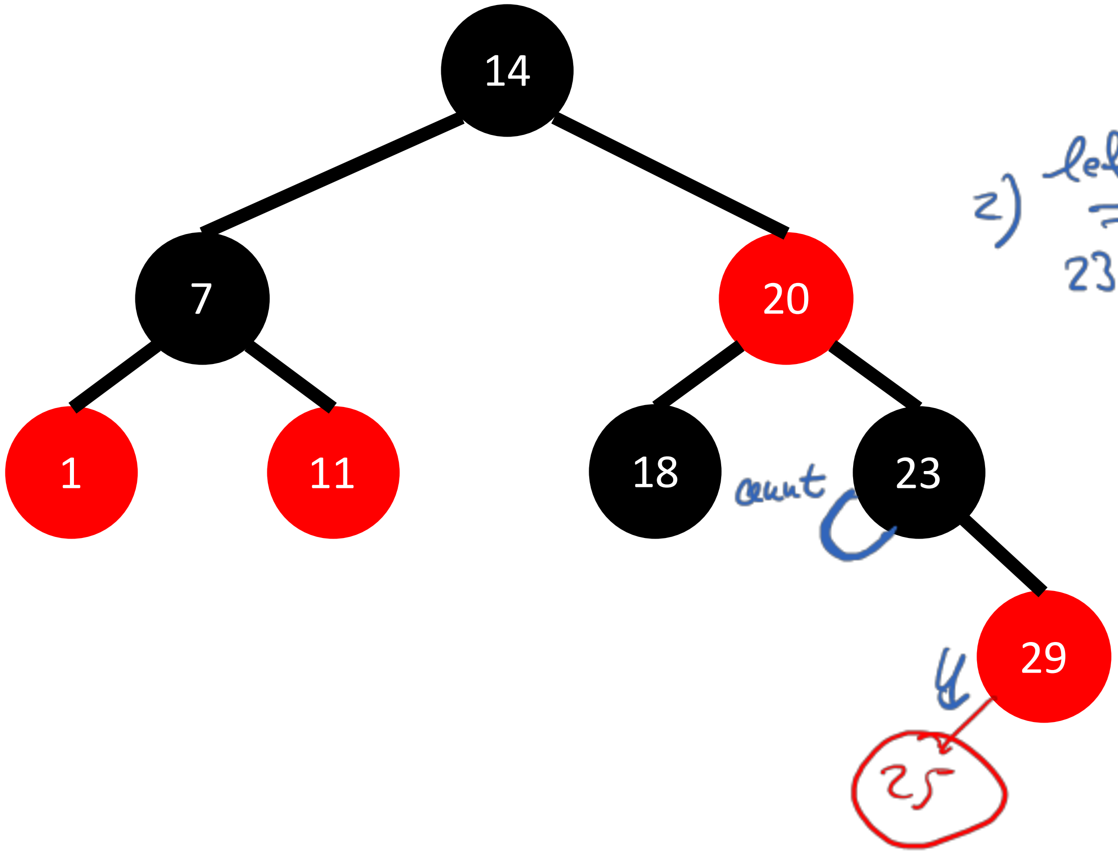
recolor

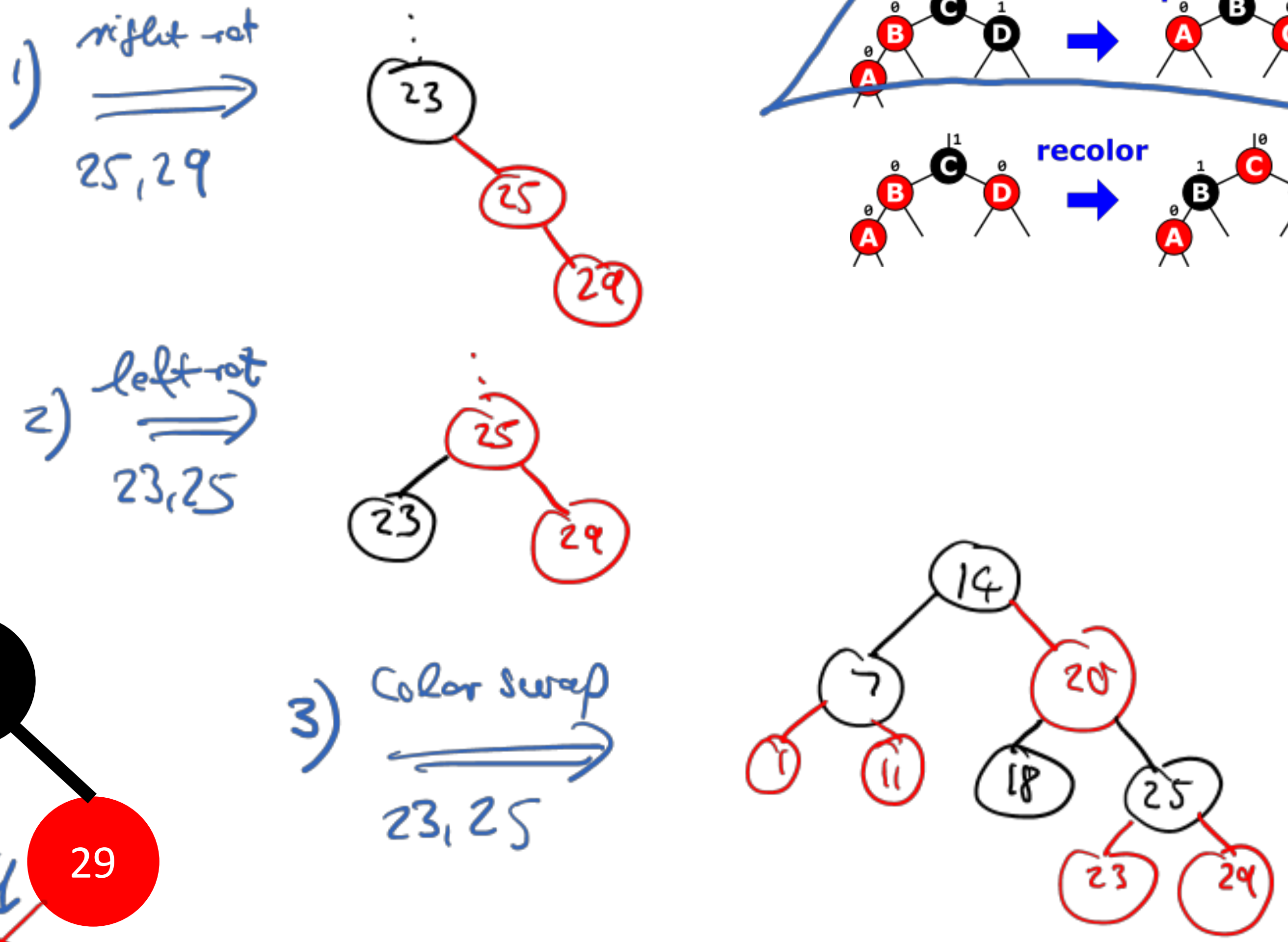


Red-Black Tree Insertion Example (7)



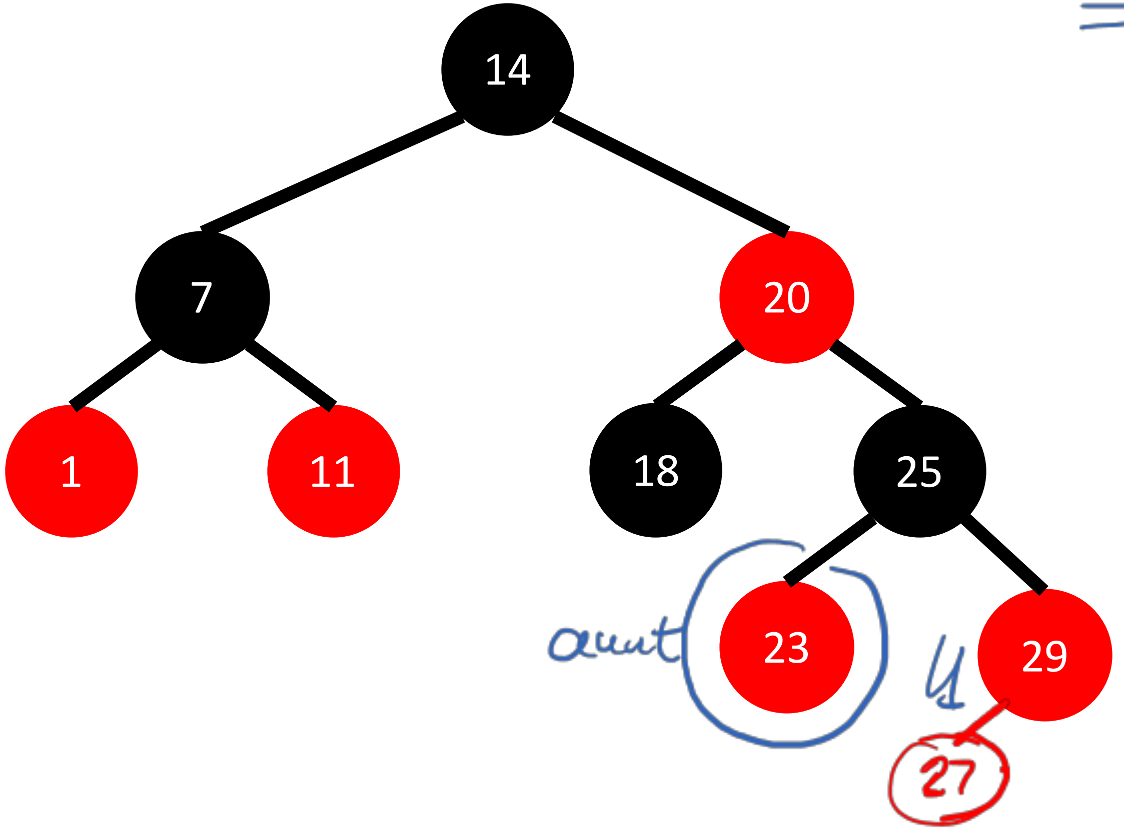
Insert: 25

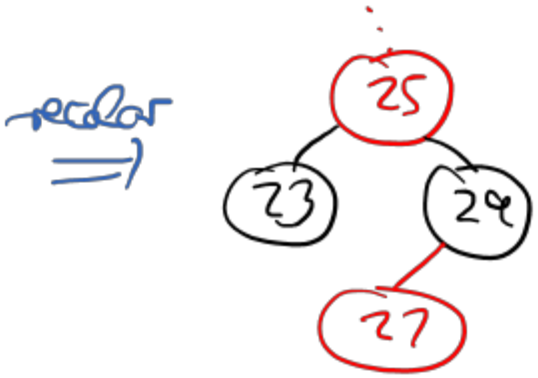


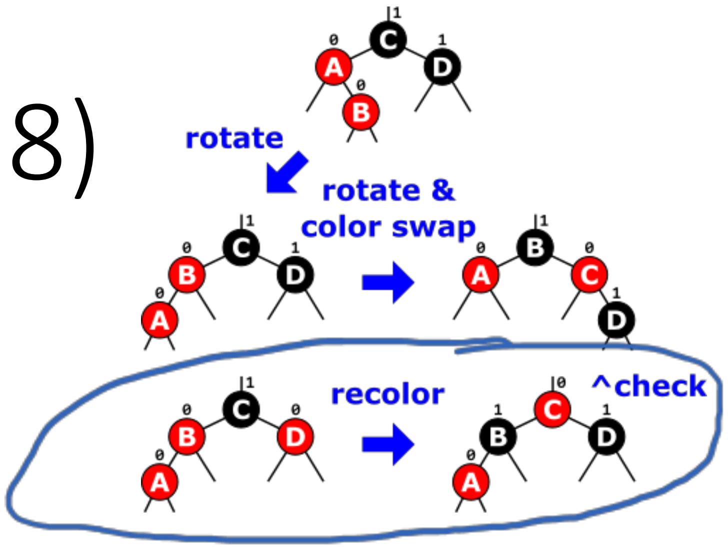


Red-Black Tree Insertion Example (8)

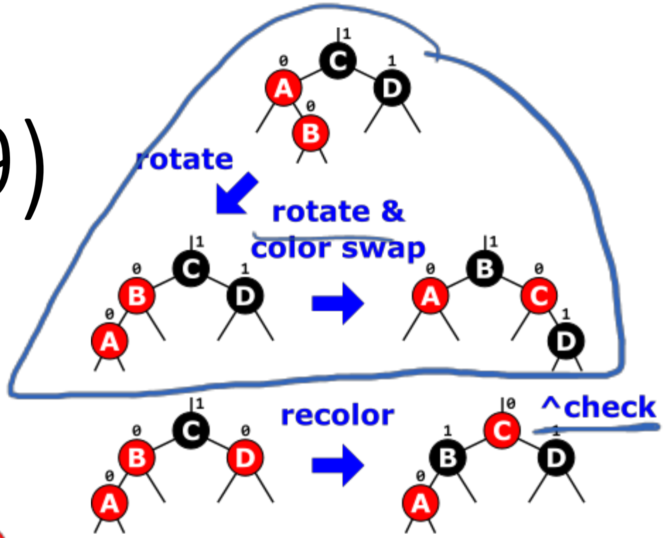
Insert: 27



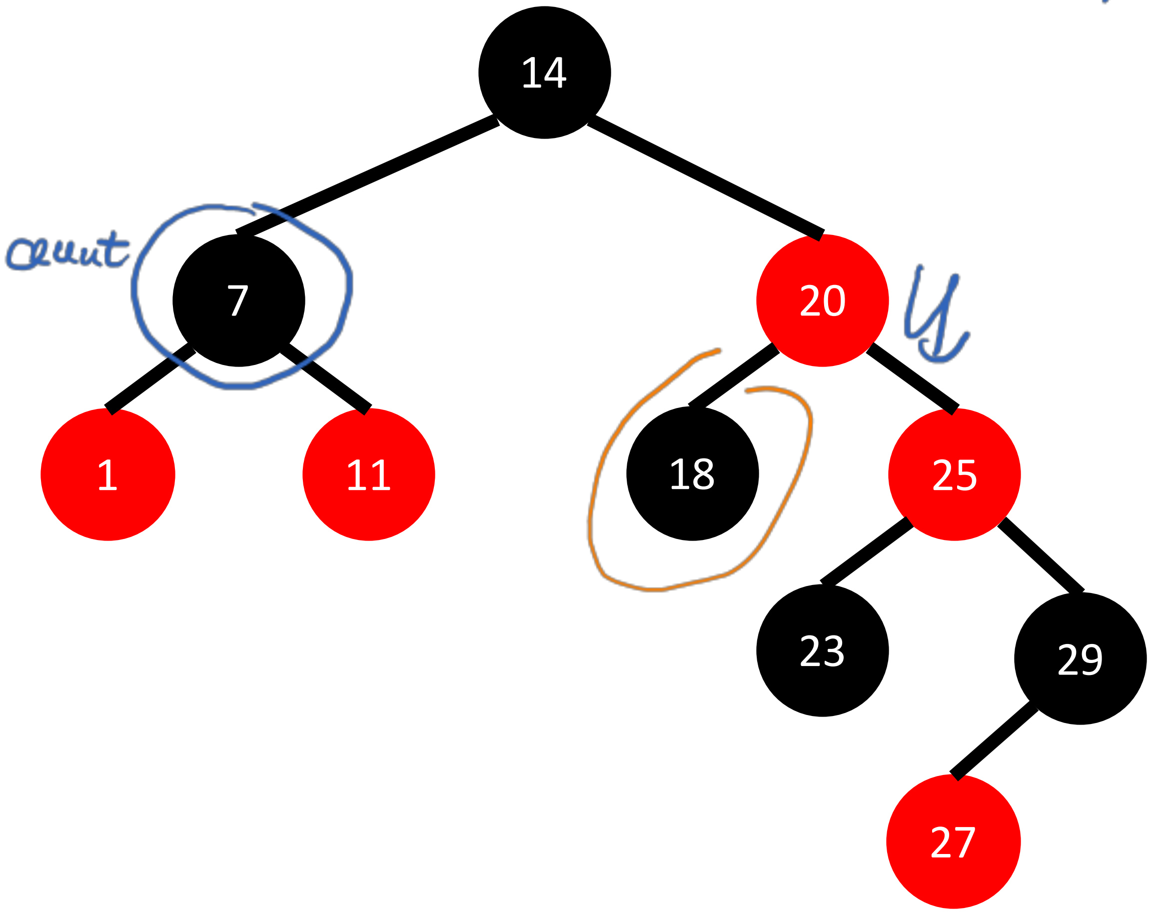


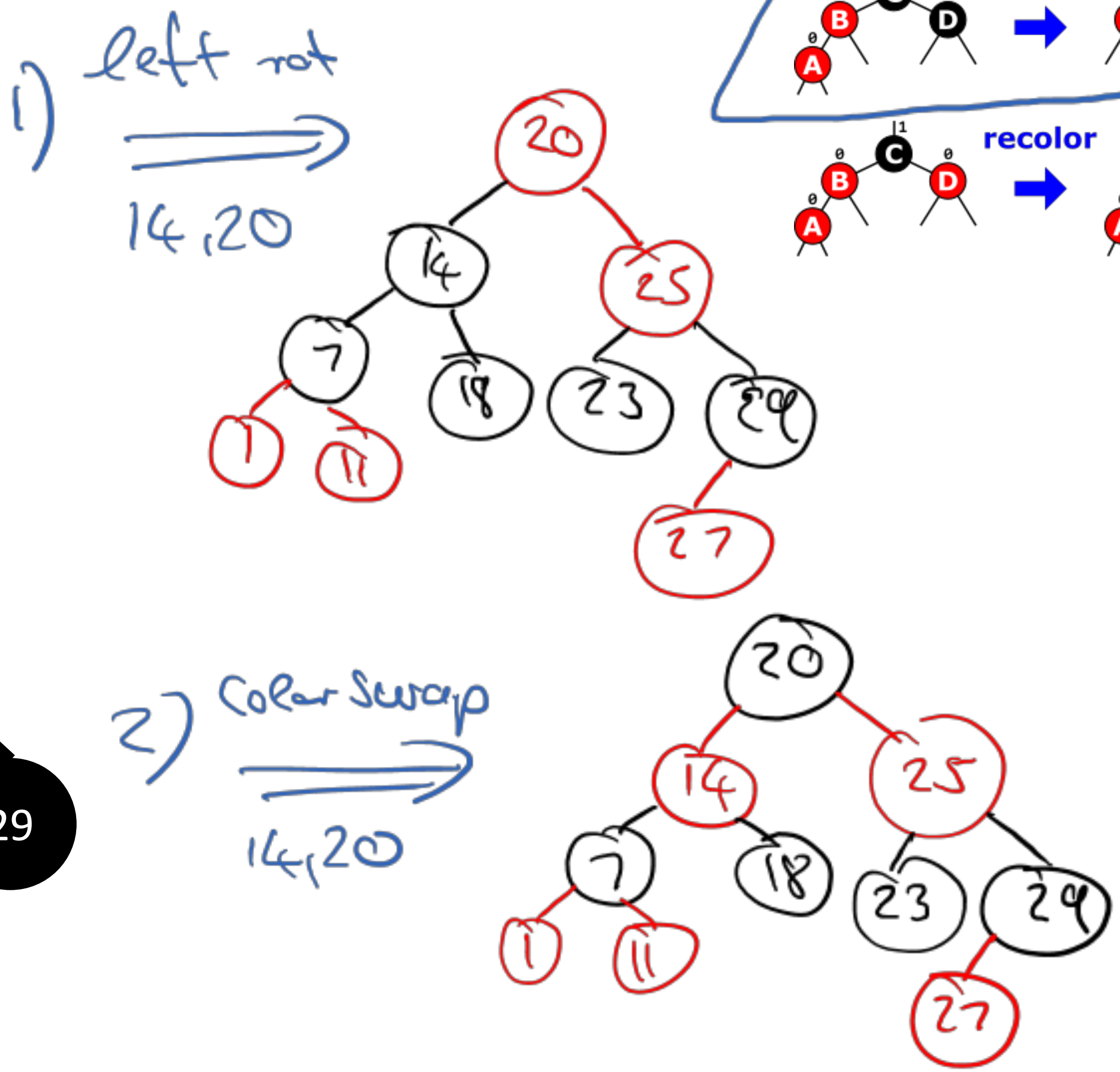


Red-Black Tree Insertion Example (9)



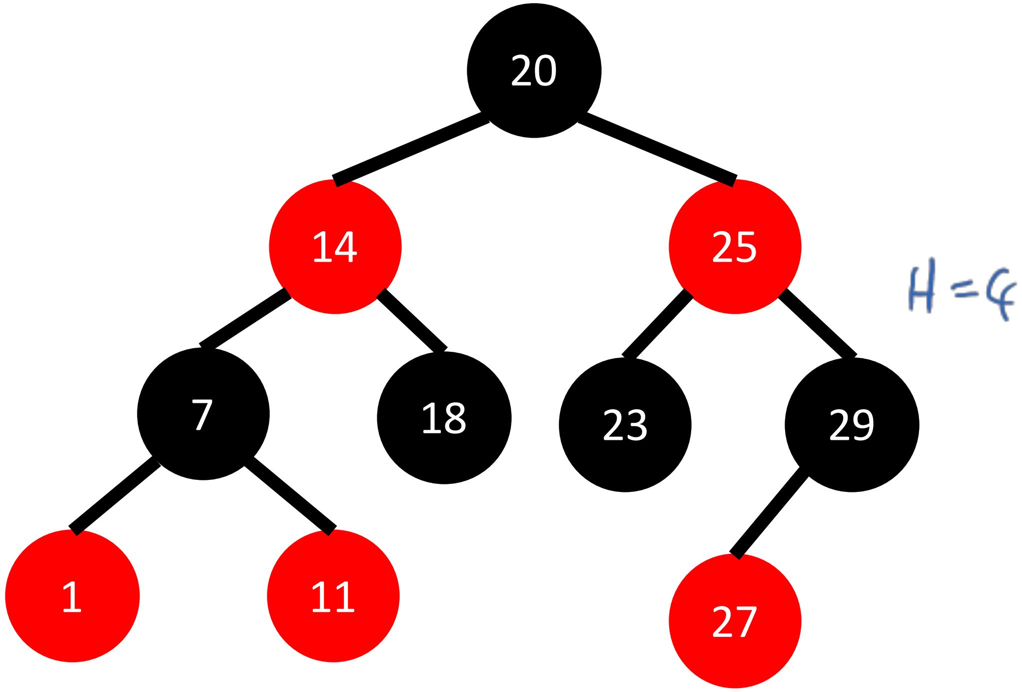
Cascading Fix

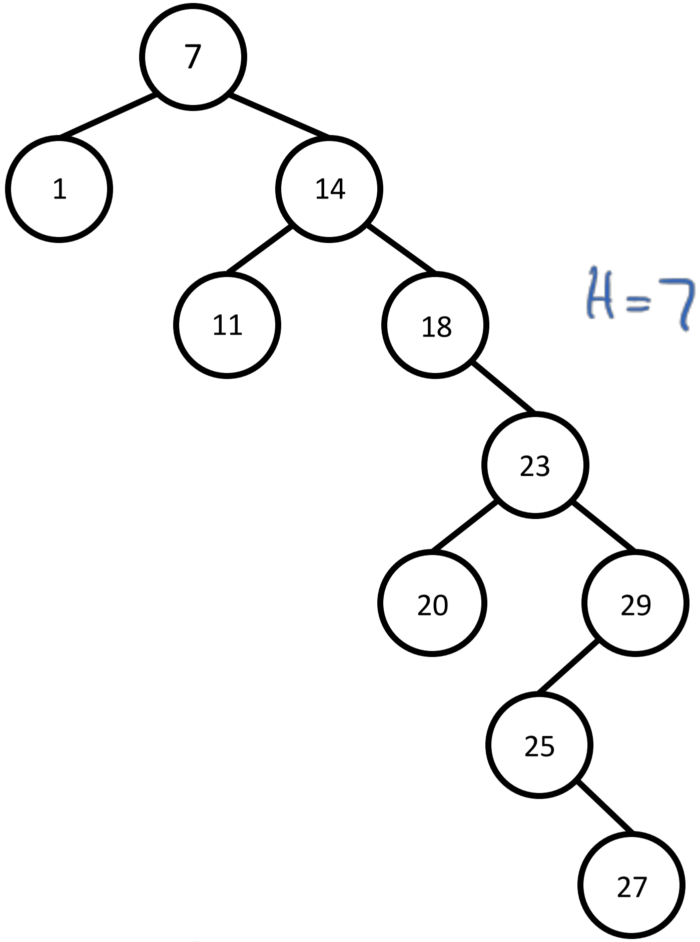




Compare to Regular BST

Insertions: 7, 14, 18, 23, 1, 11, 20, 29, 25, 27



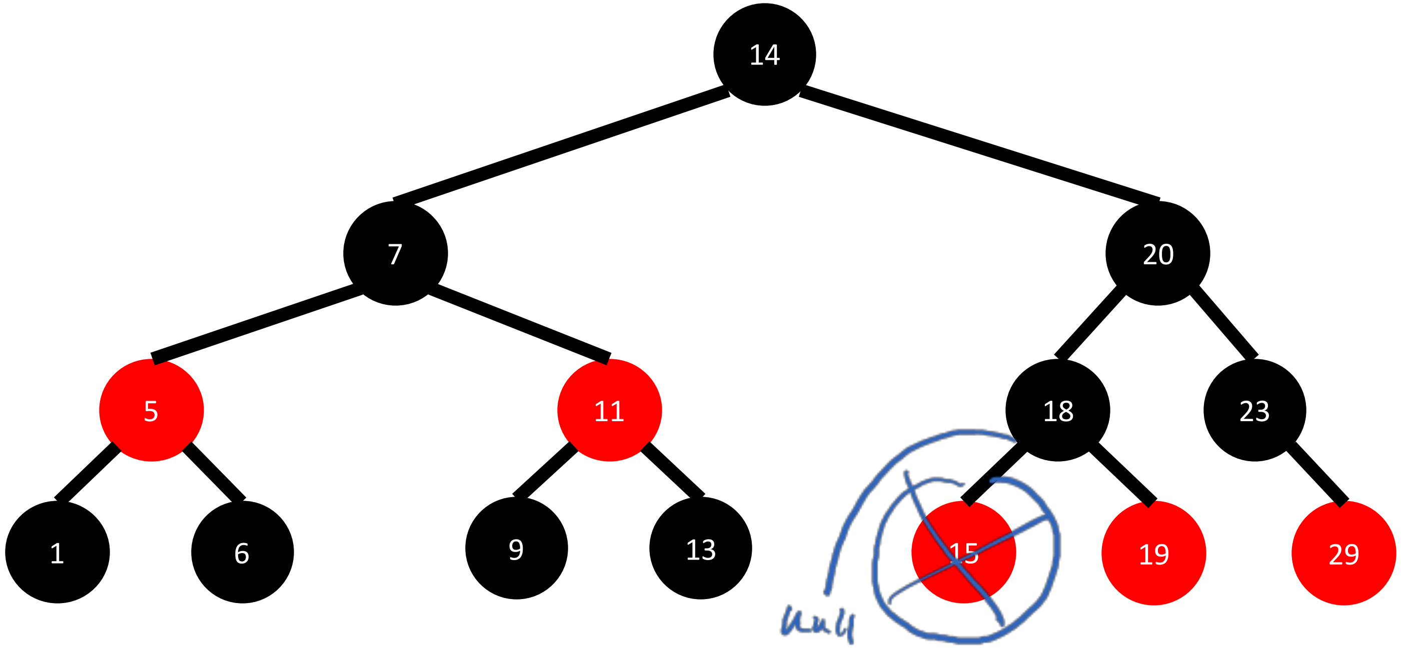


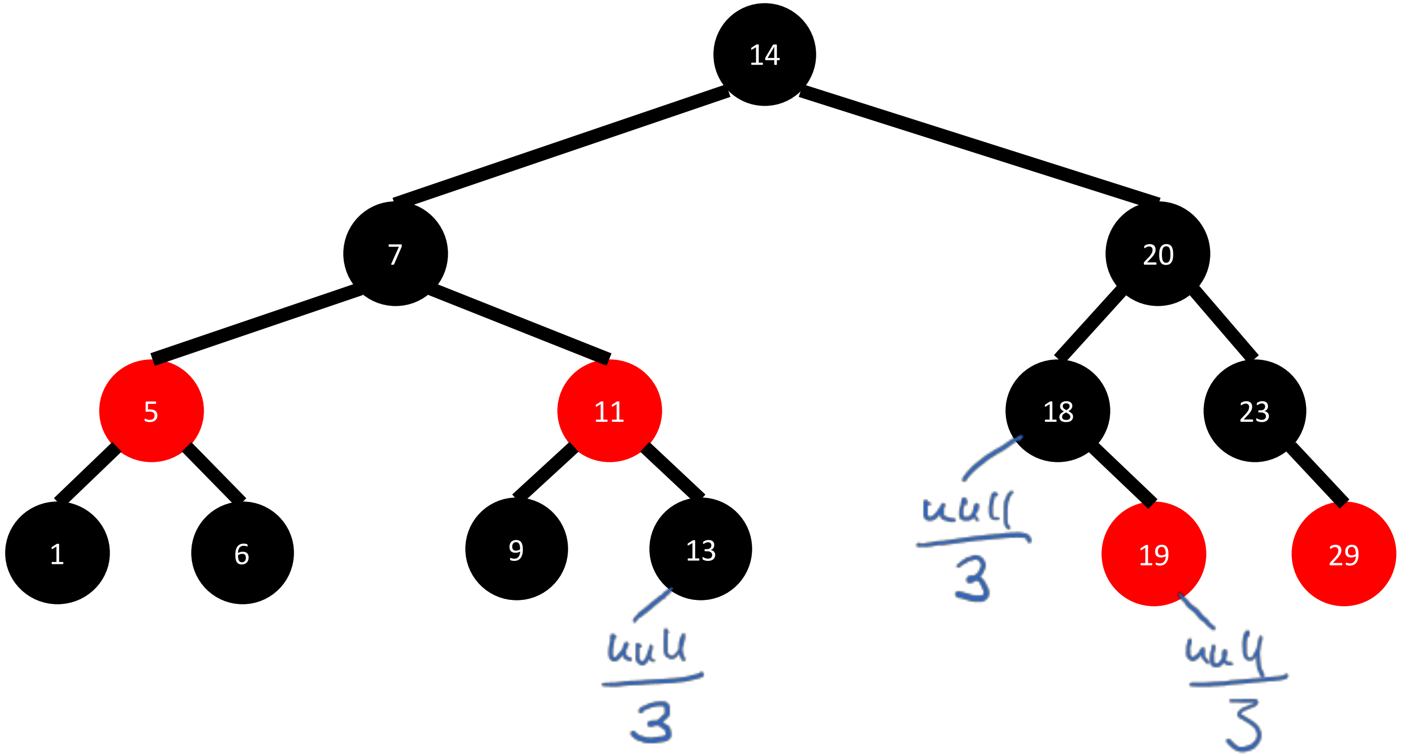
Red-Black Tree Deletion

Deleting from a Red-Black Tree

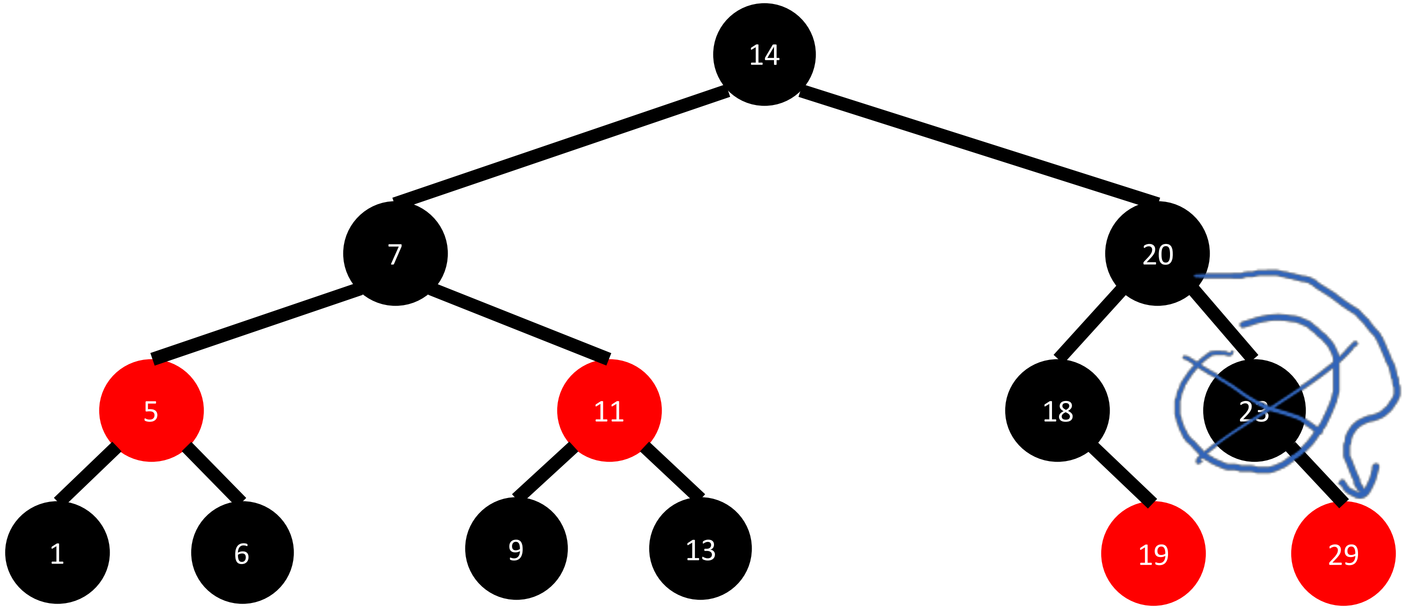
1. Delete value using the BST deletion algorithm (3 cases)
2. Check Red-Black tree properties and restore if necessary

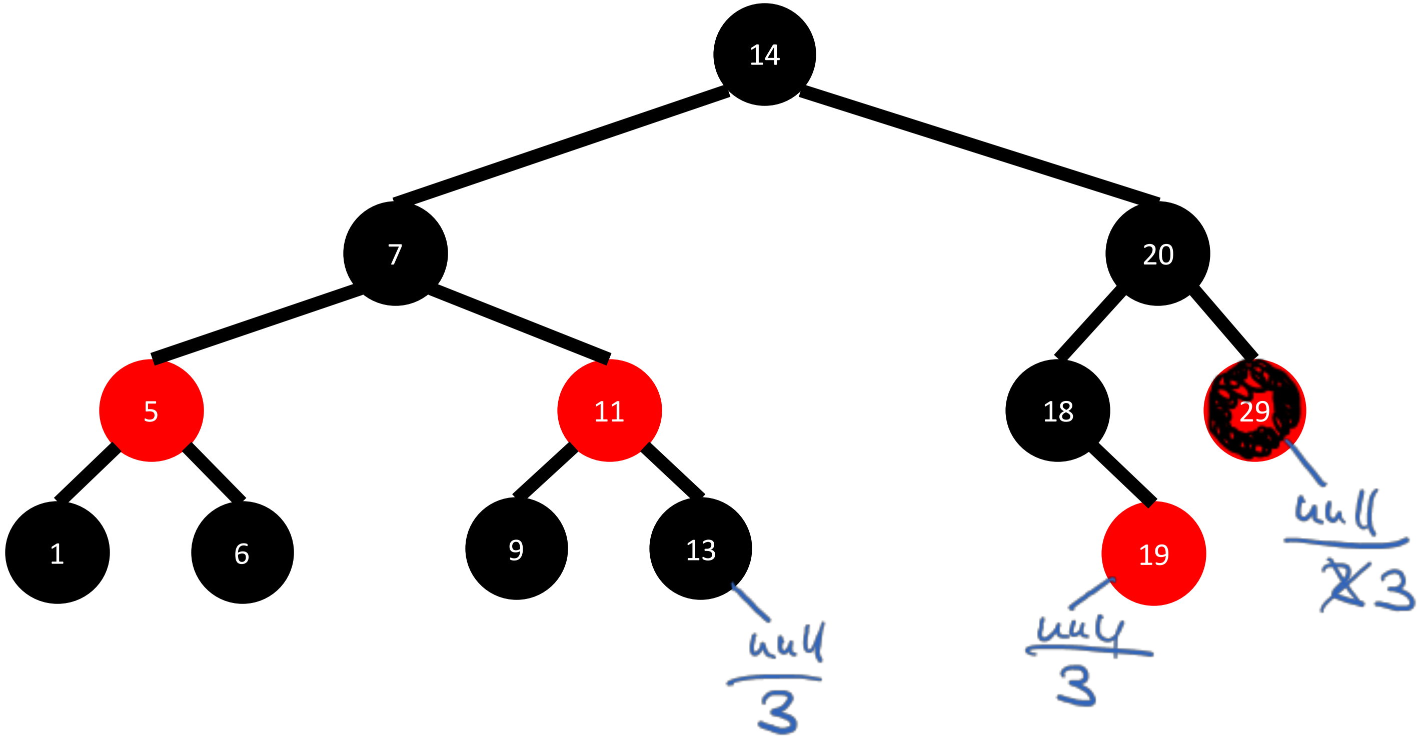
Delete 15



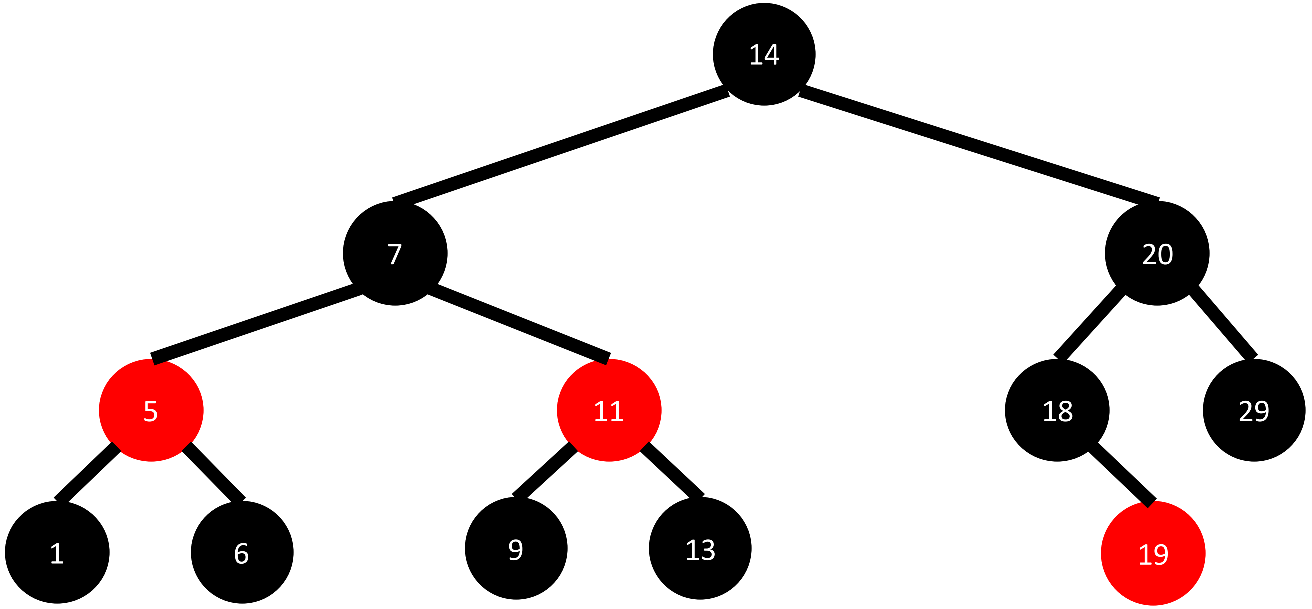

14
7
20
5
11
18
23
1
6
9
13
19
29
null
3
null
3
null
3

Delete 23

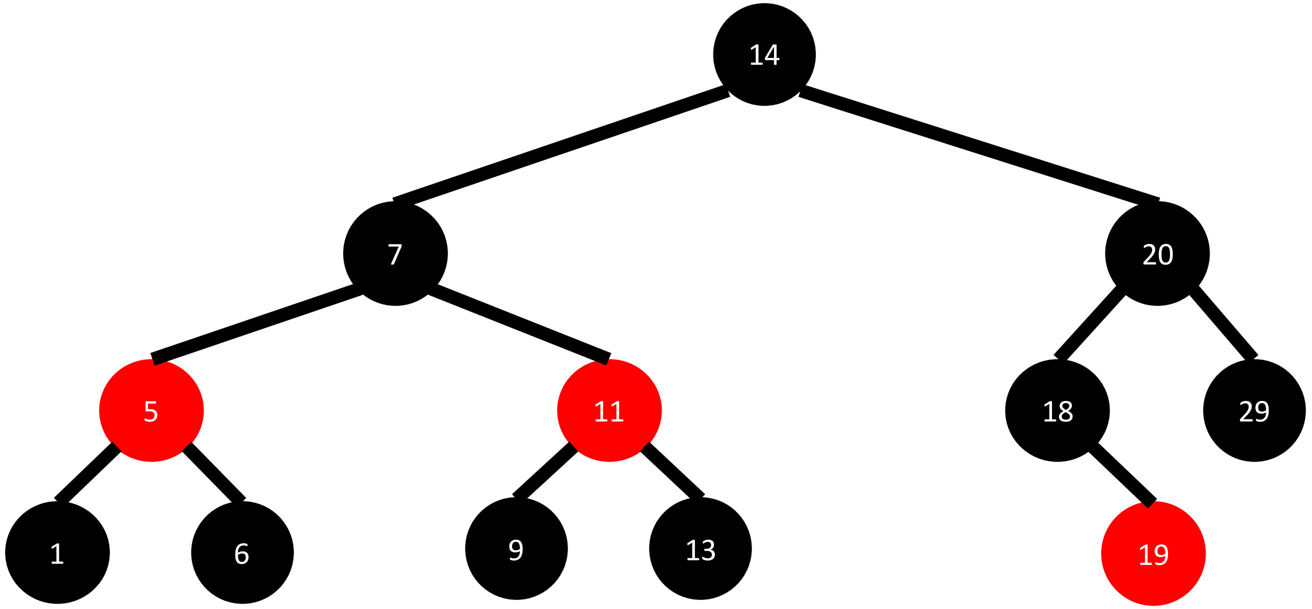




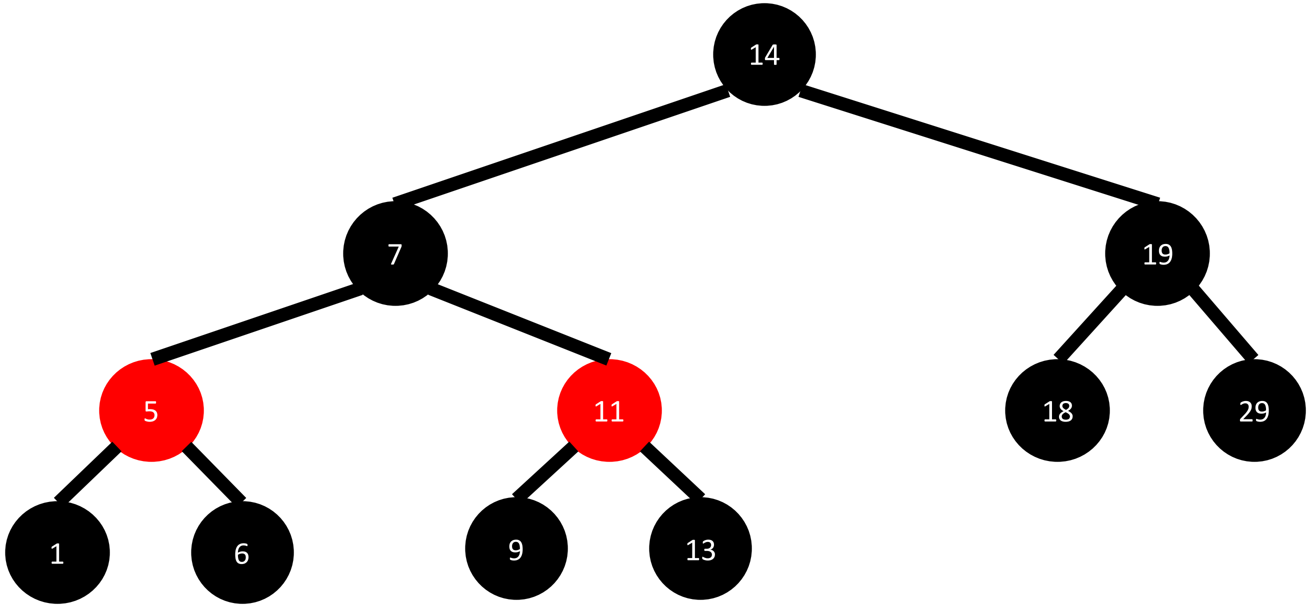
14
7
20
5
11
18
29
1
6
9
13
19
null
3
null
3
null
~~2~~3


14
7
20
5
11
18
29
1
6
9
13
19

Delete 20



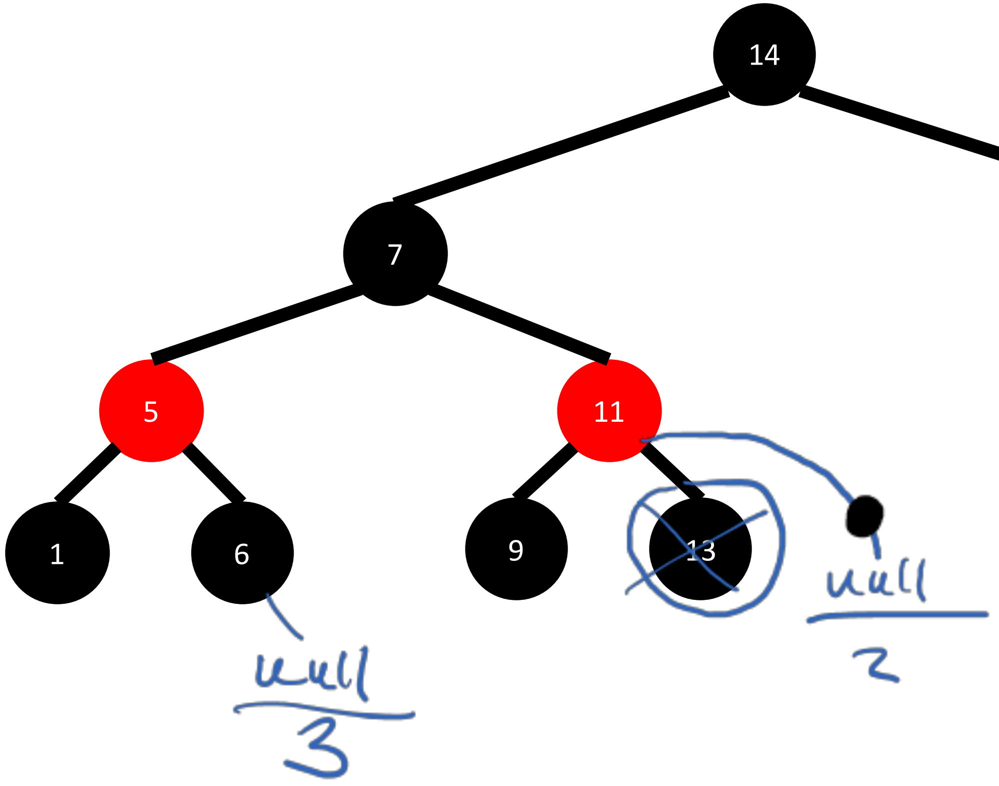
Deleted Node or Replacement Node is RED



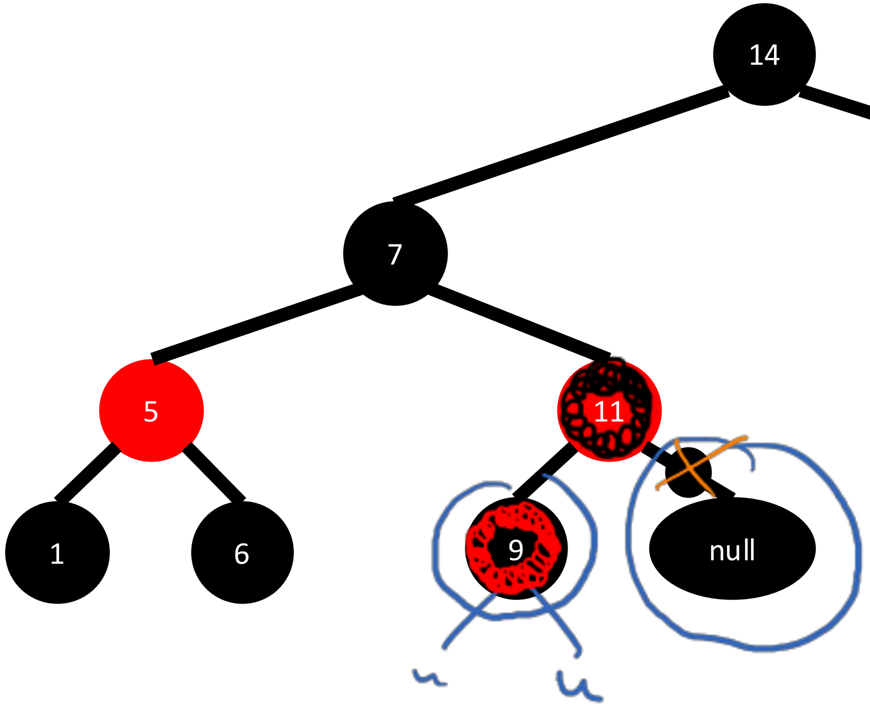
If deleted node is **BLACK** and replacement is **RED**, turn replacement **BLACK**.

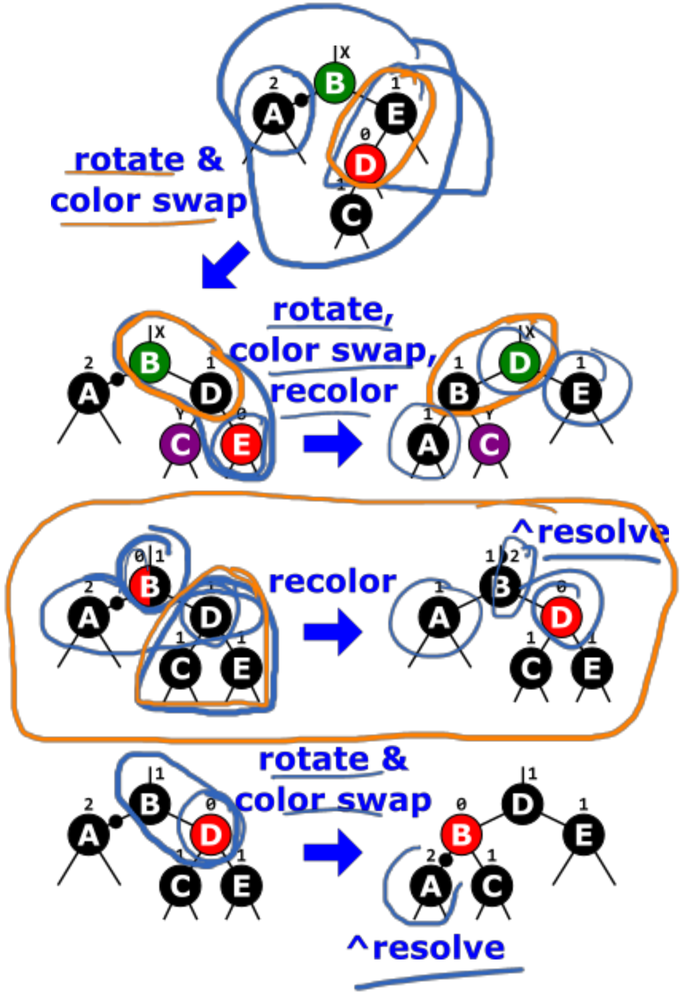
BLACK Node Without RED Replacement

Delete: 13



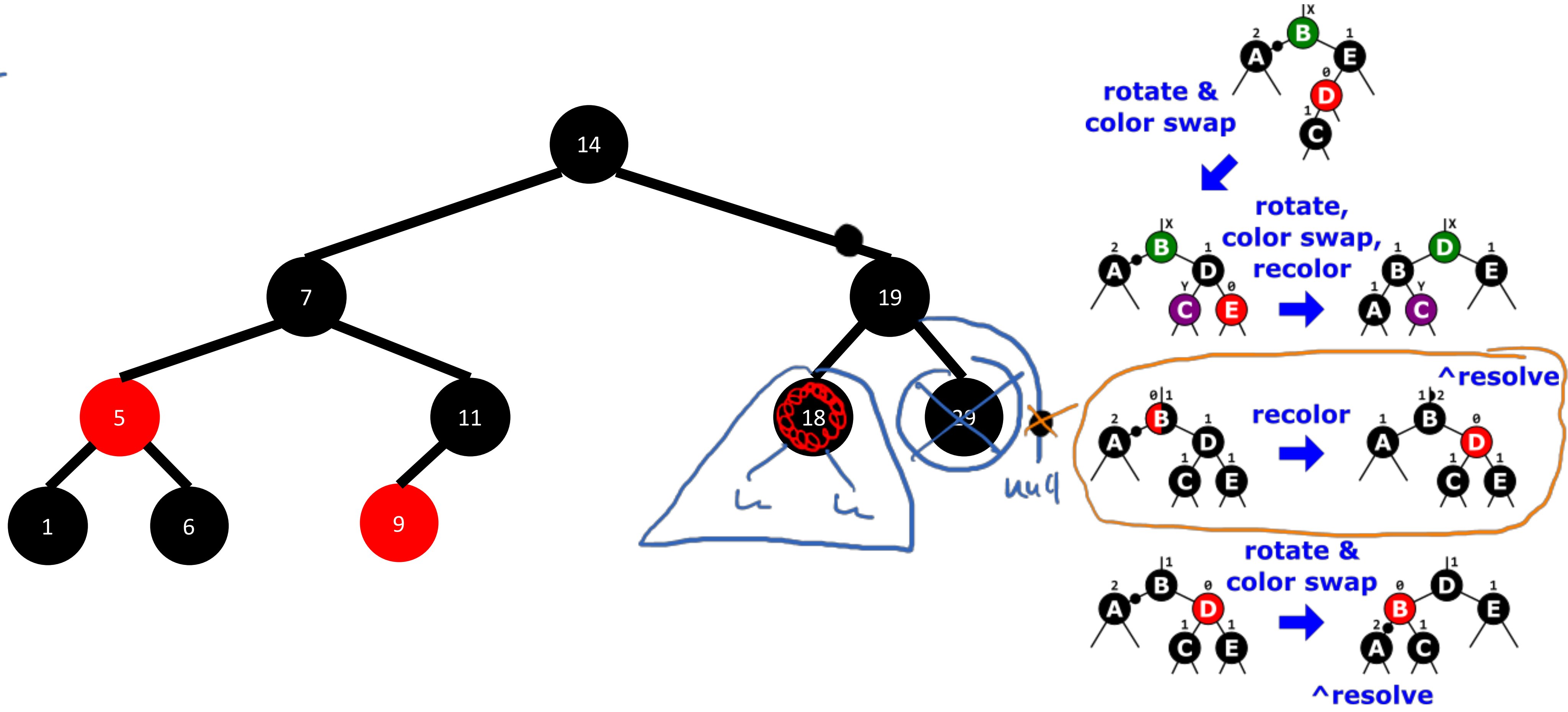
BLACK Node Without RED Replacement



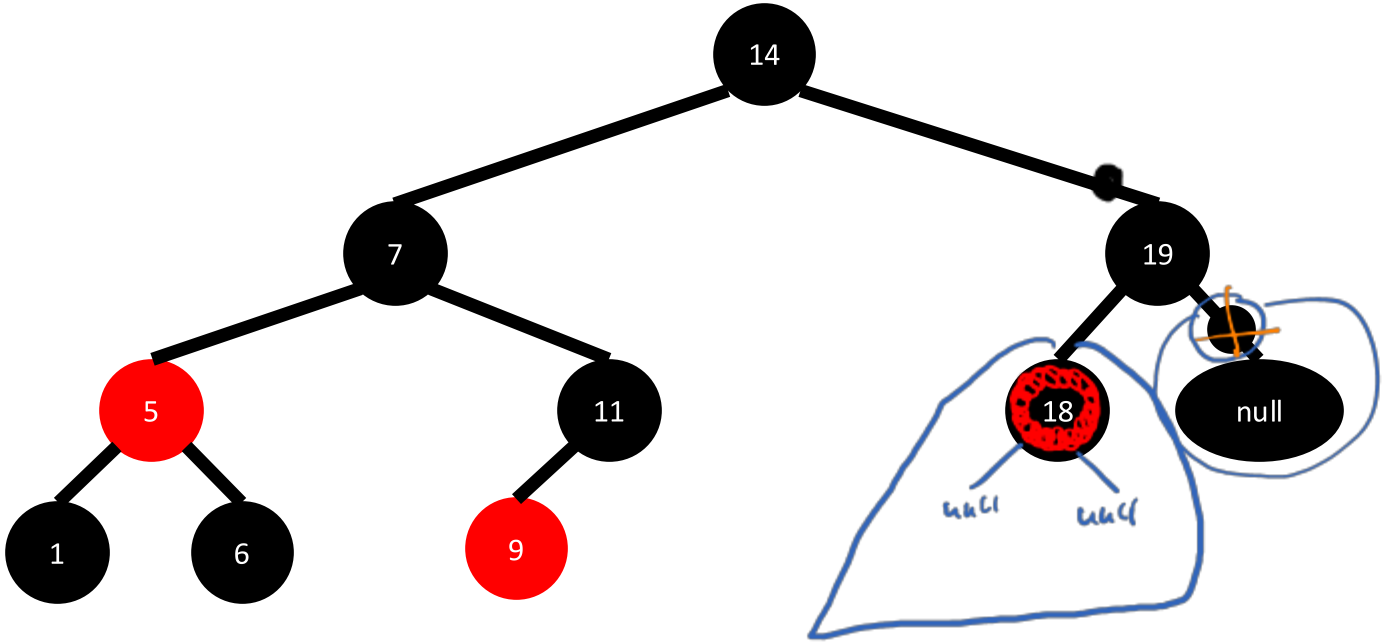


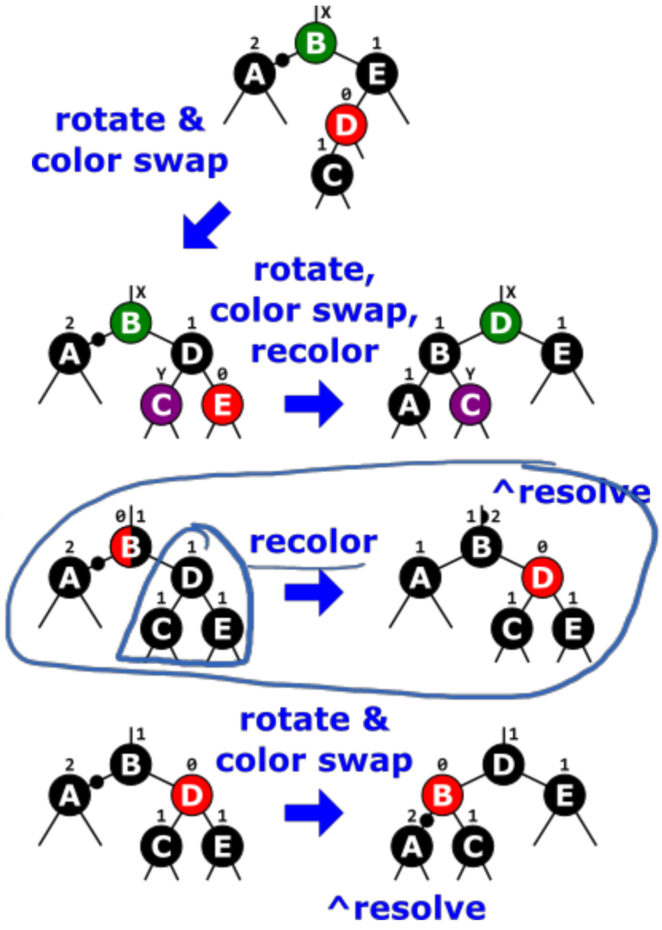
BLACK Node Without RED Replacement

Delete: 29

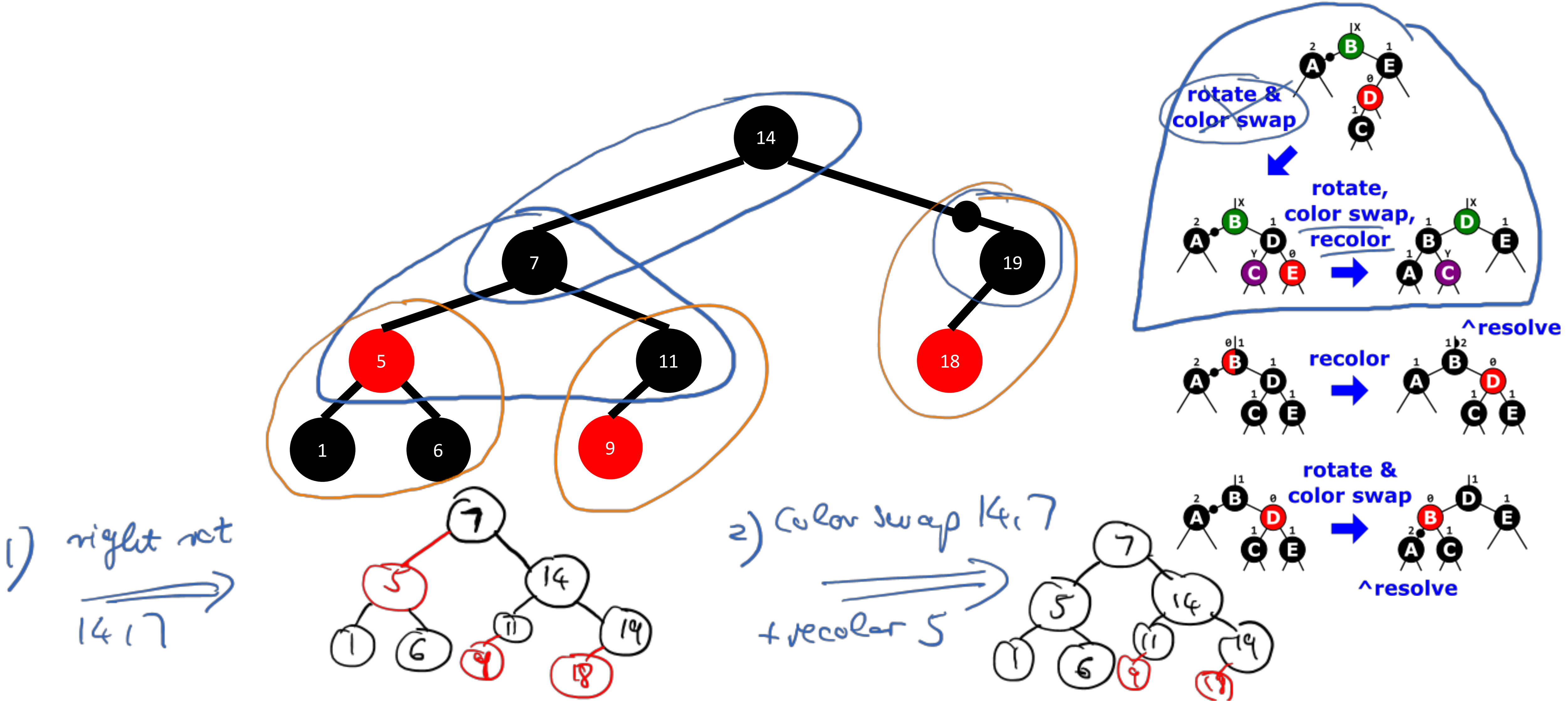


BLACK Node Without RED Replacement



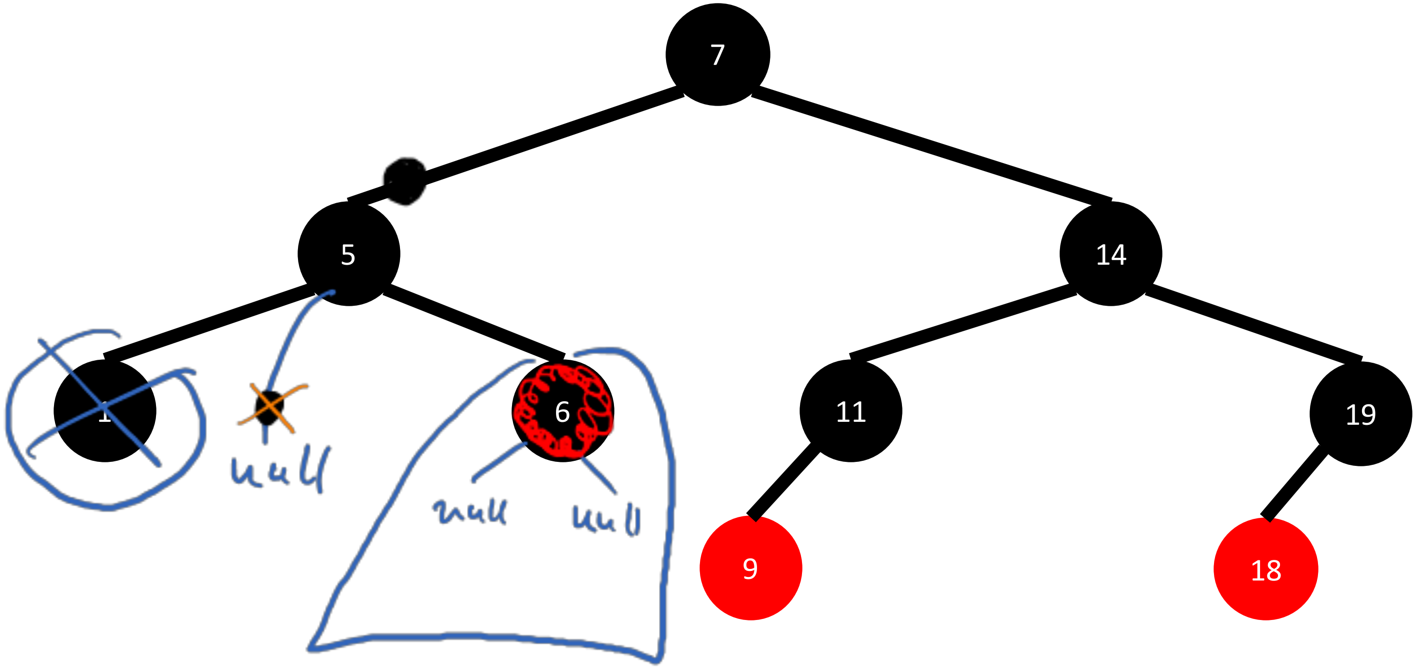


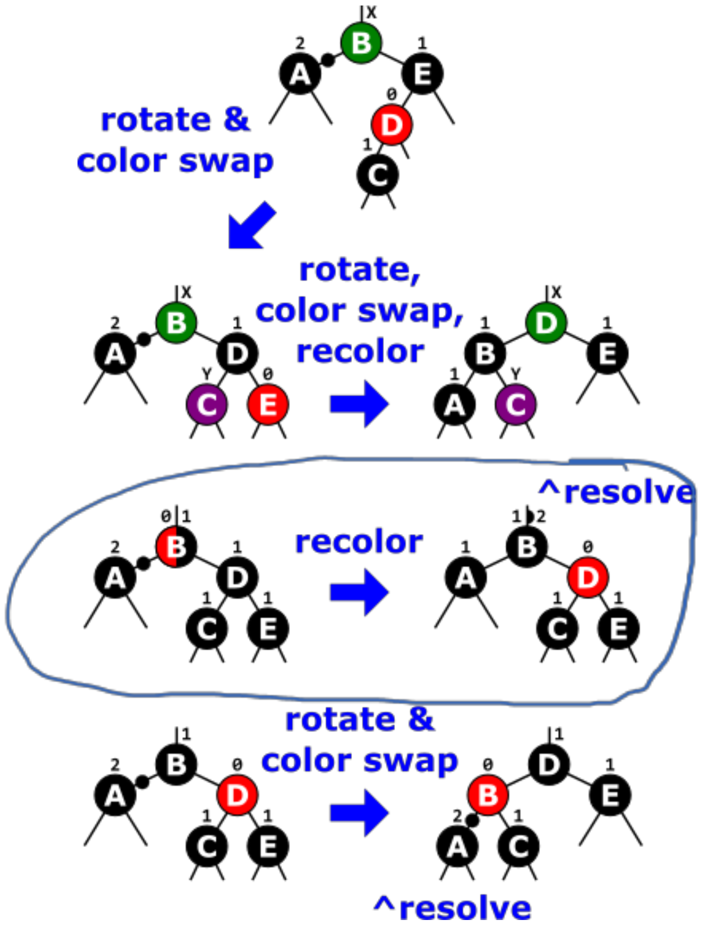
BLACK Node Without RED Replacement



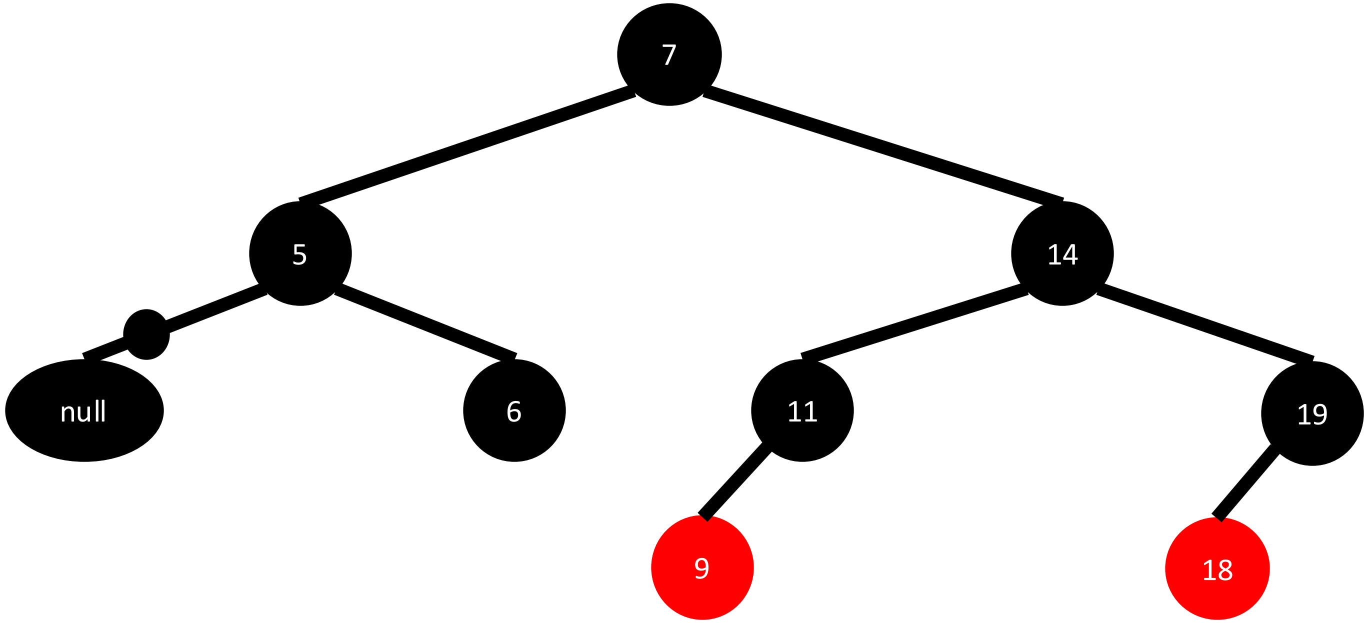
BLACK Node Without RED Replacement

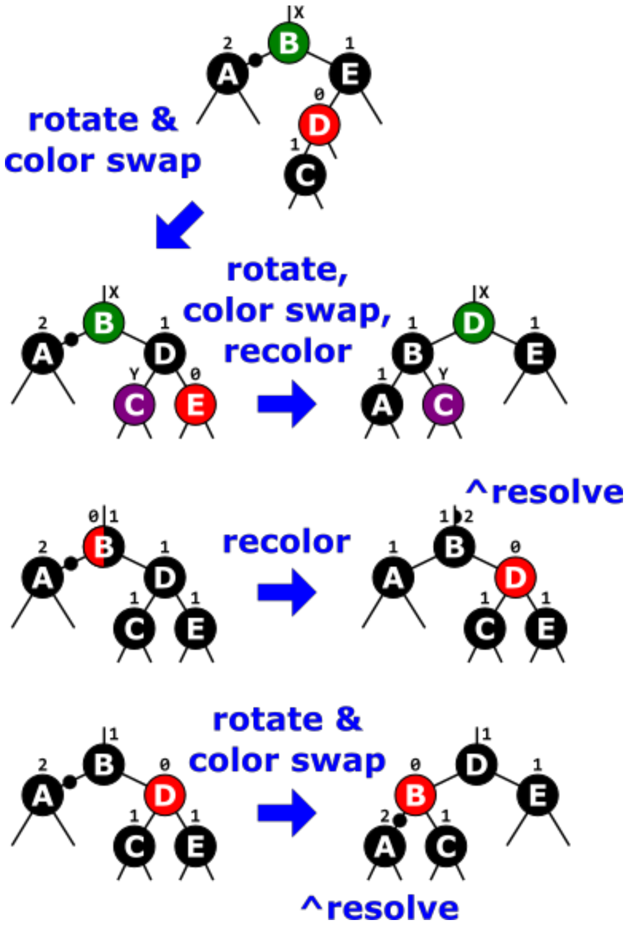
Delete: 1



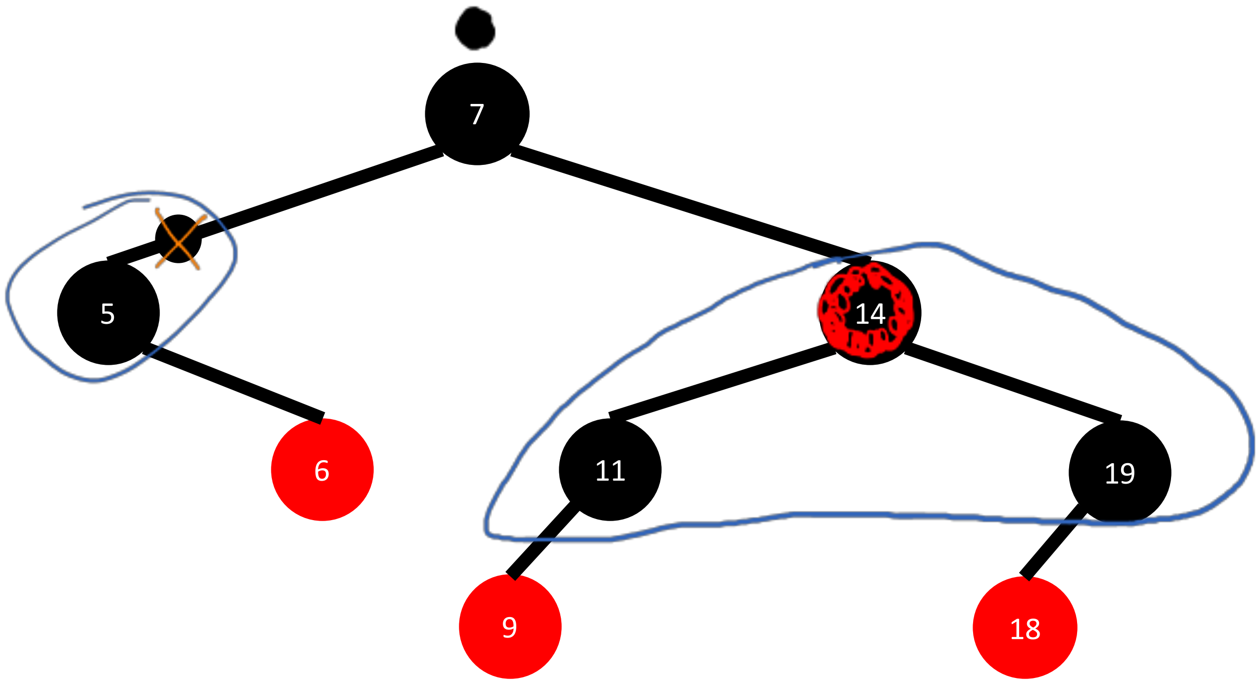


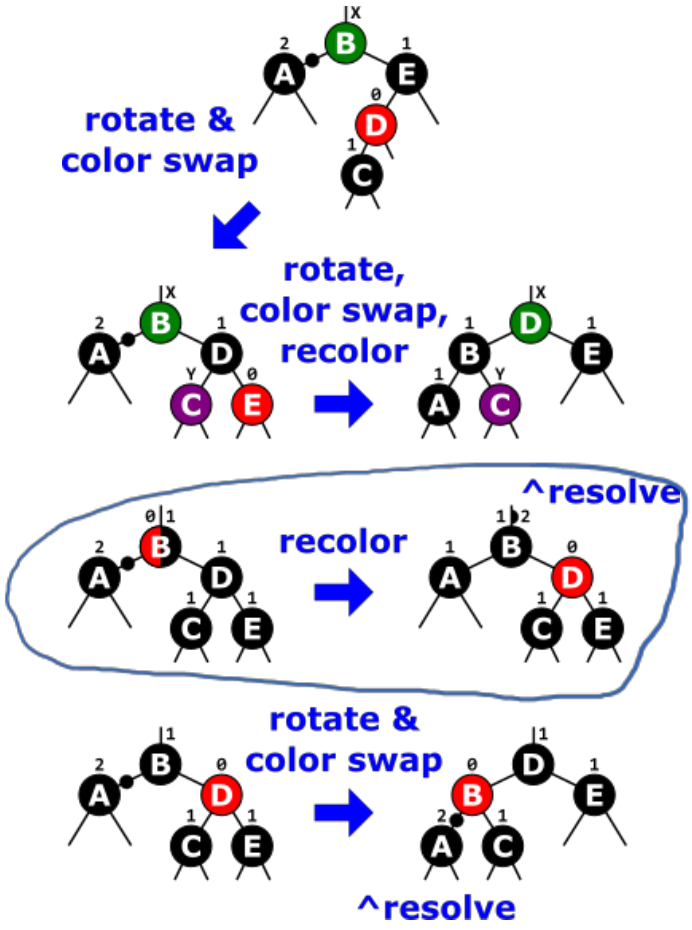
BLACK Node Without RED Replacement



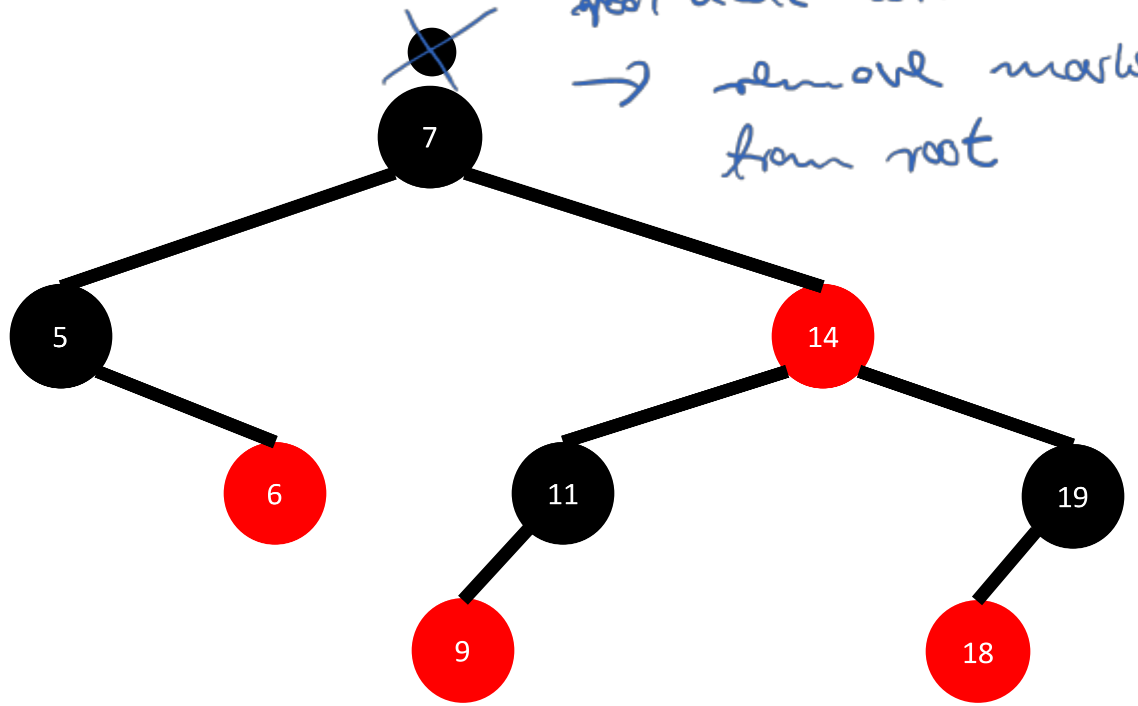


BLACK Node Without RED Replacement

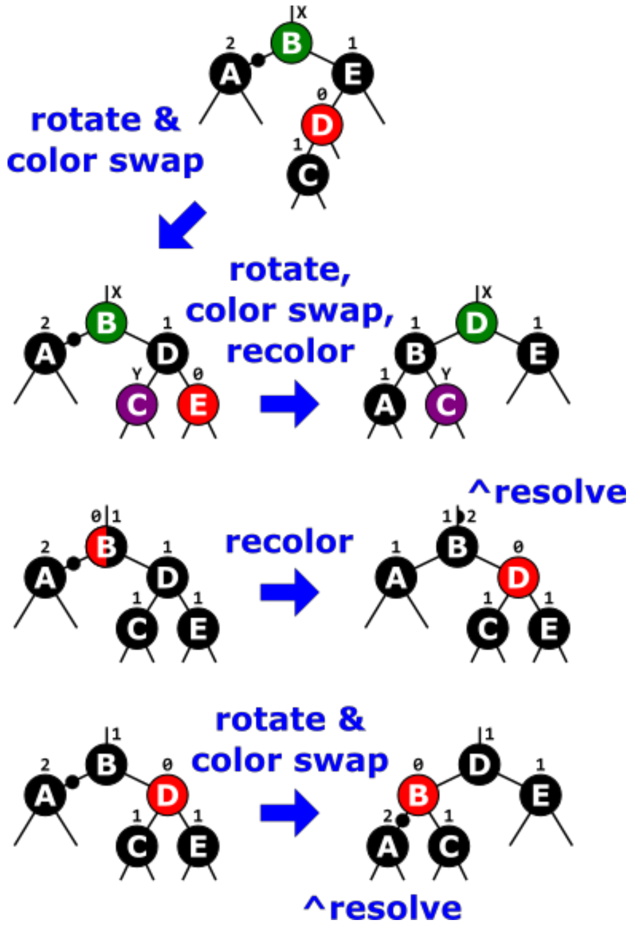




BLACK Node Without RED Replacement

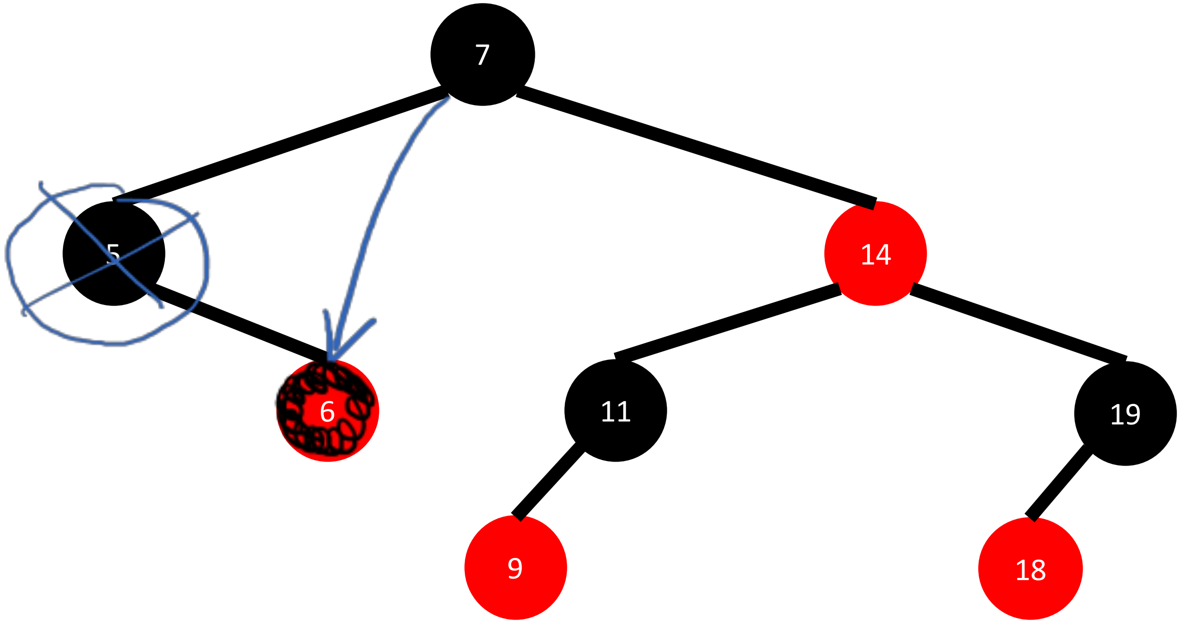


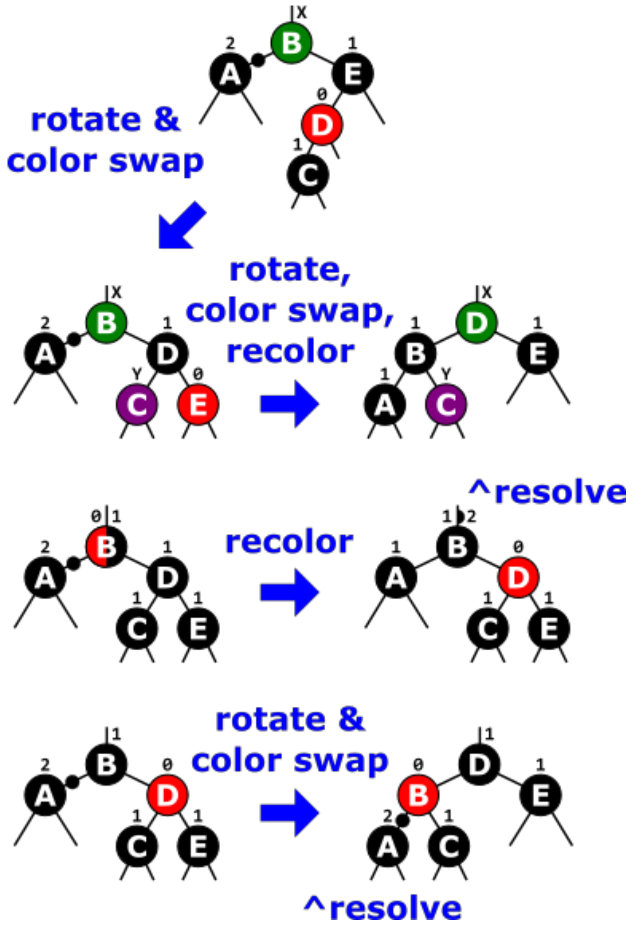
root node with marker
→ remove marker
from root



BLACK Node With RED Replacement

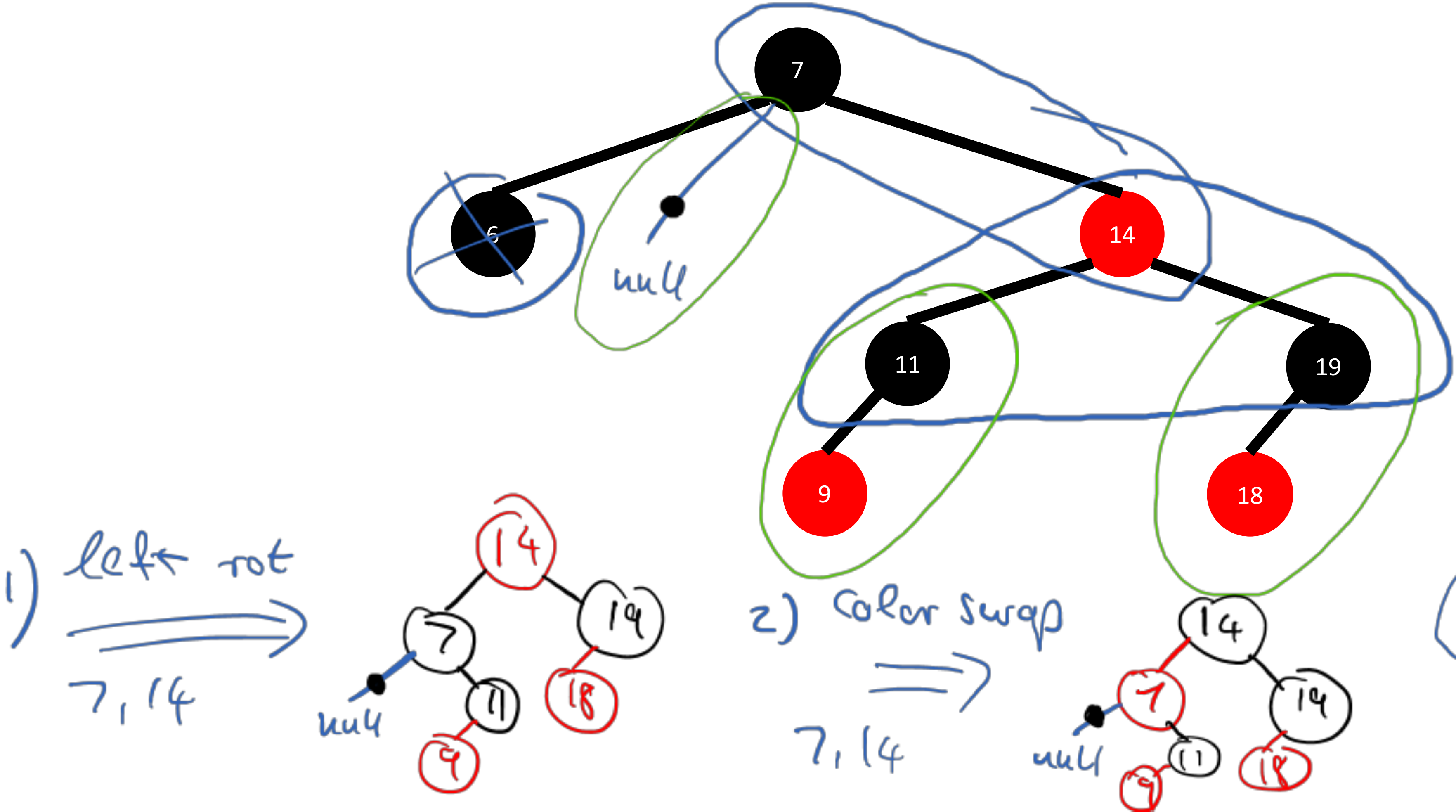
Delete: 5

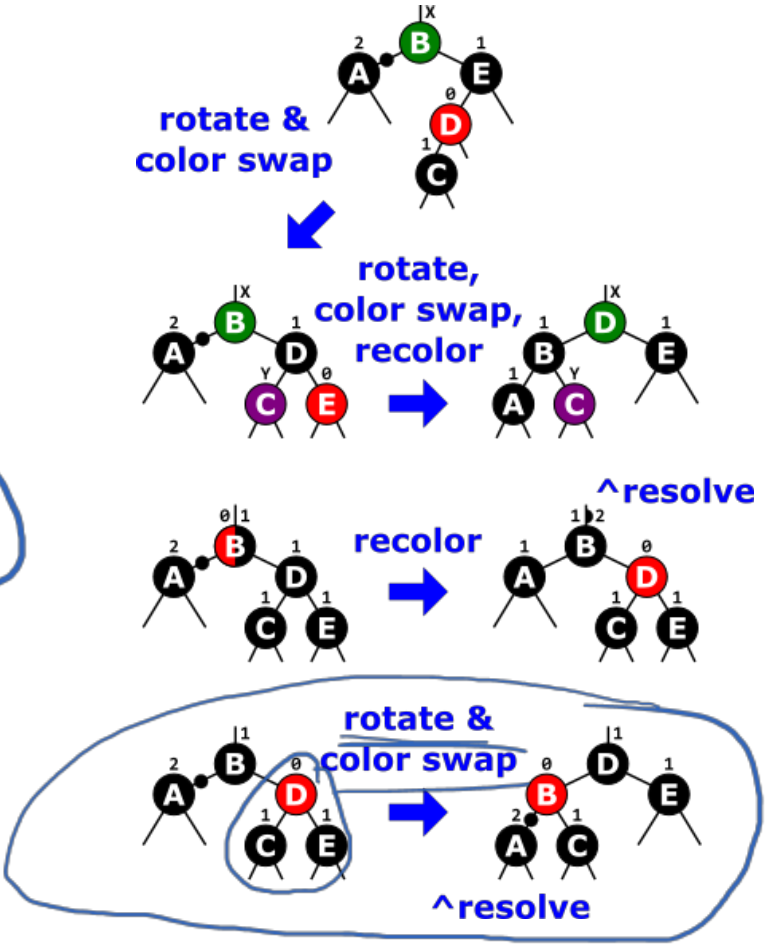




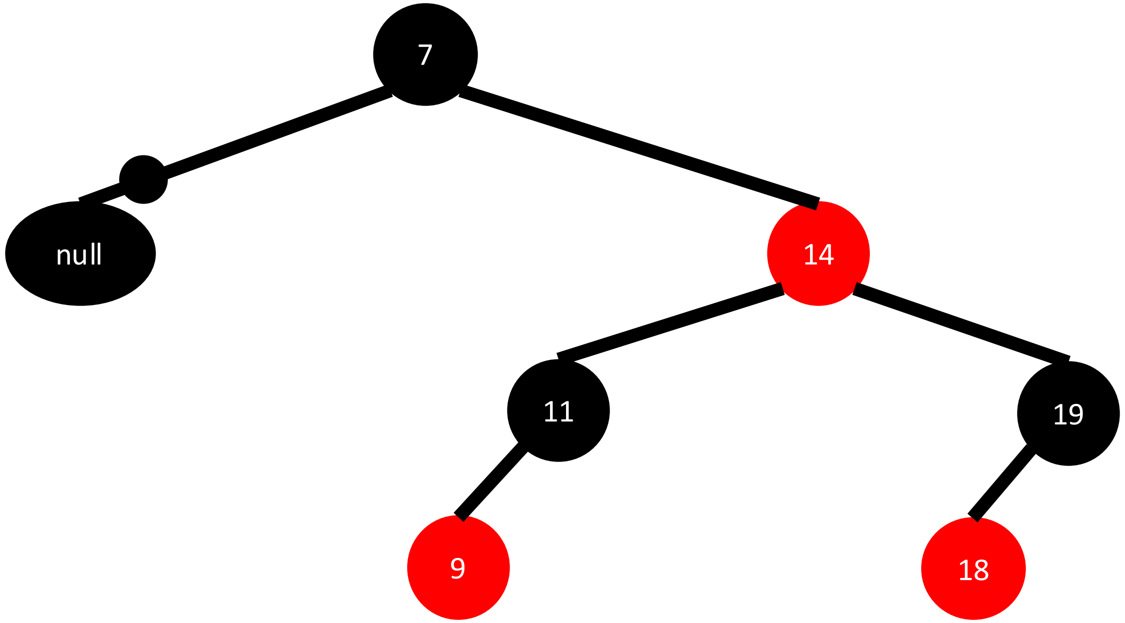
BLACK Node Without RED Replacement

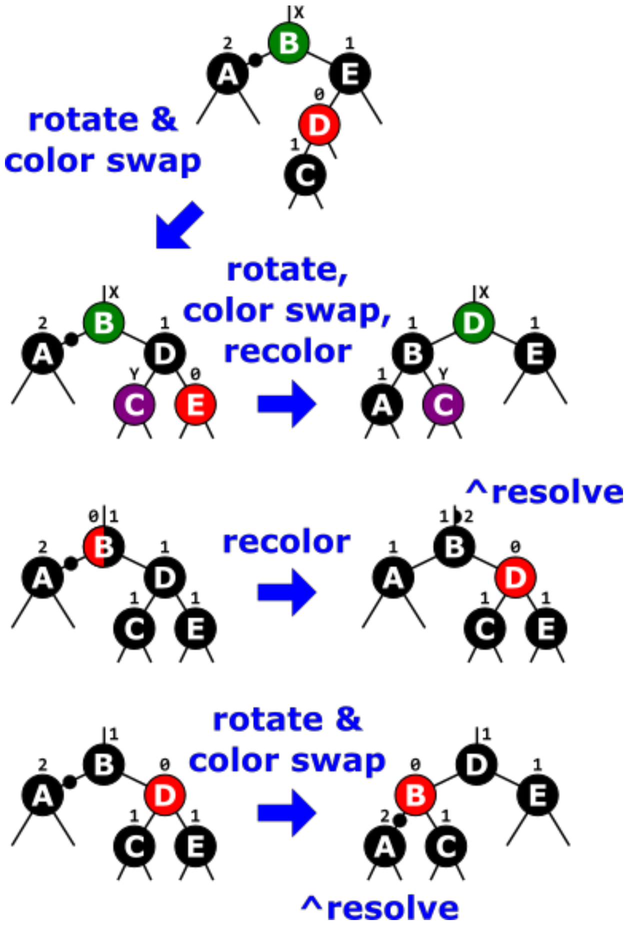
Delete: 6



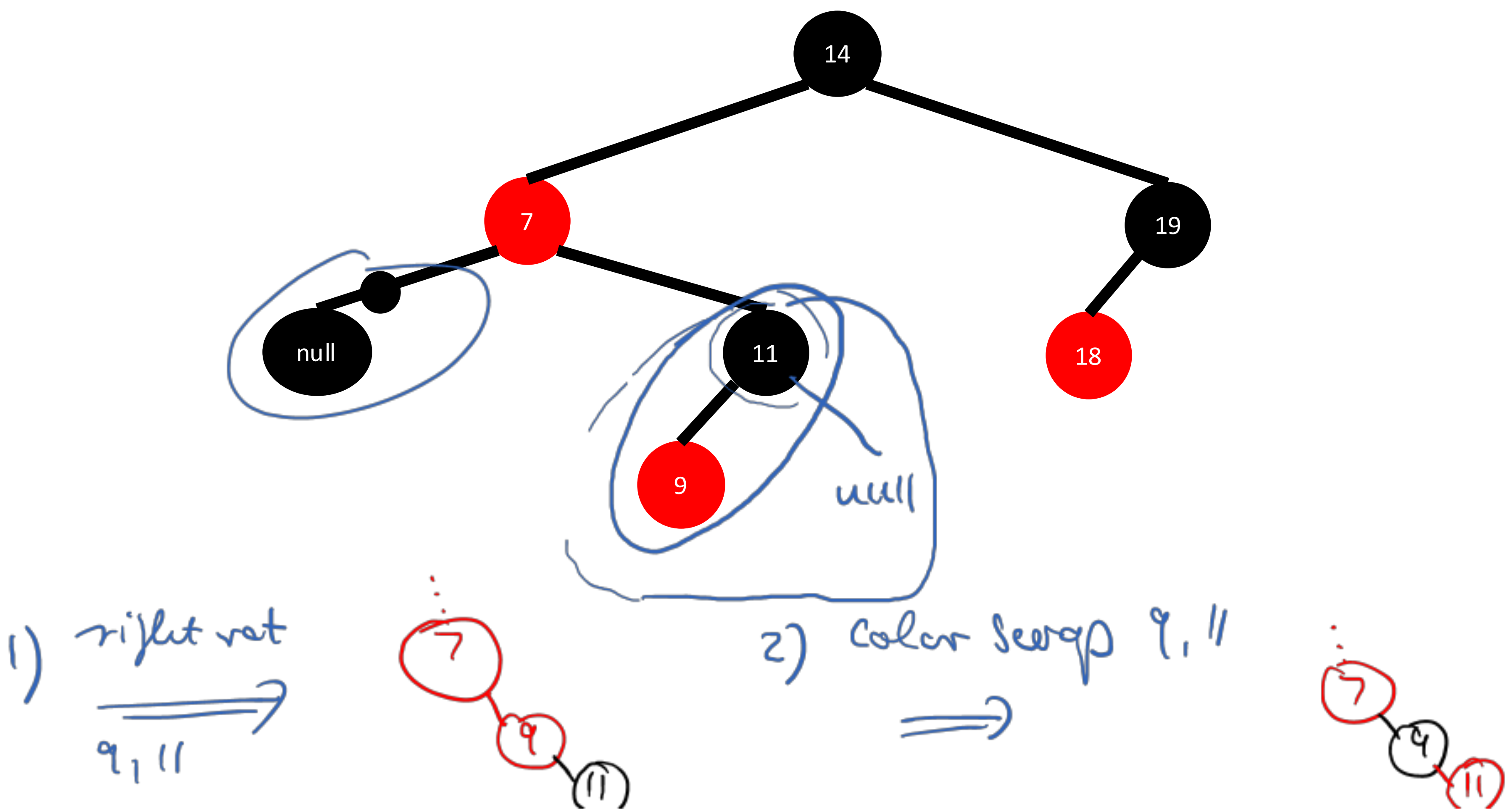


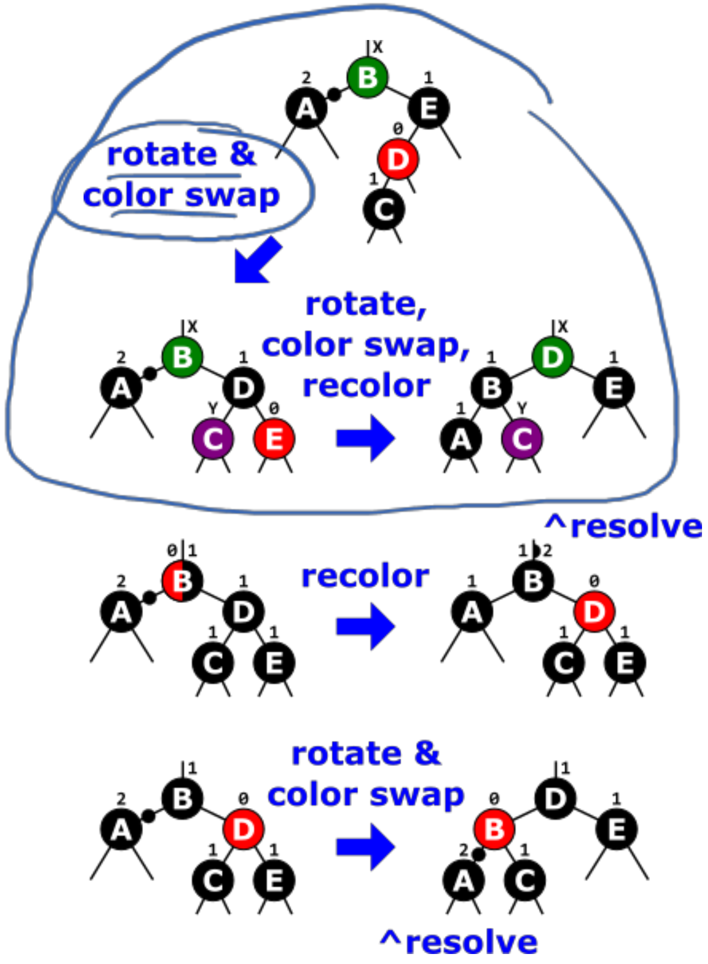
BLACK Node Without RED Replacement



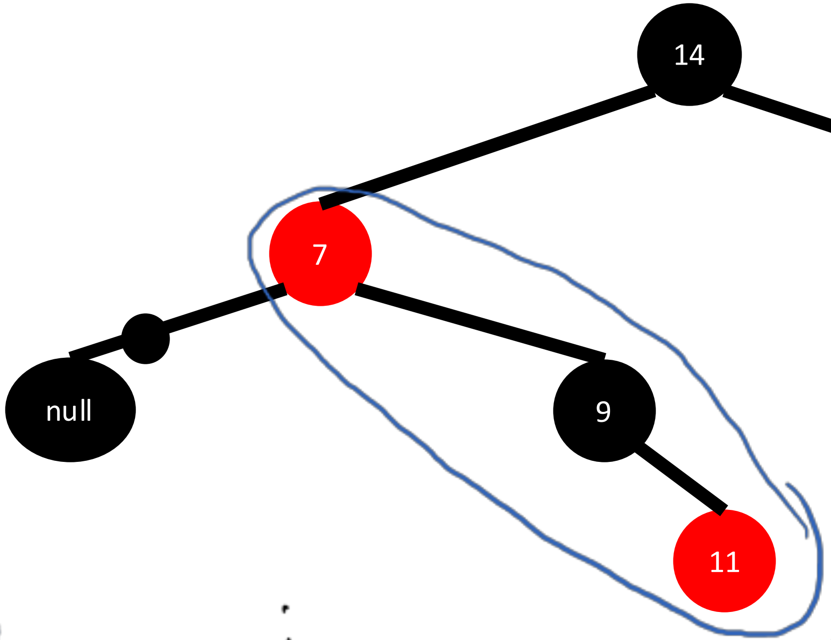


BLACK Node Without RED Replacement

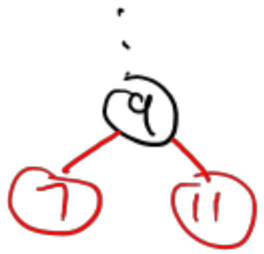




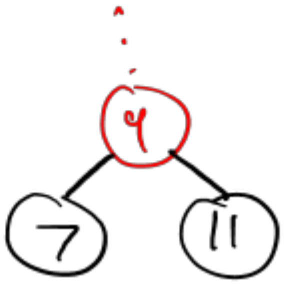
BLACK Node Without RED Replacement

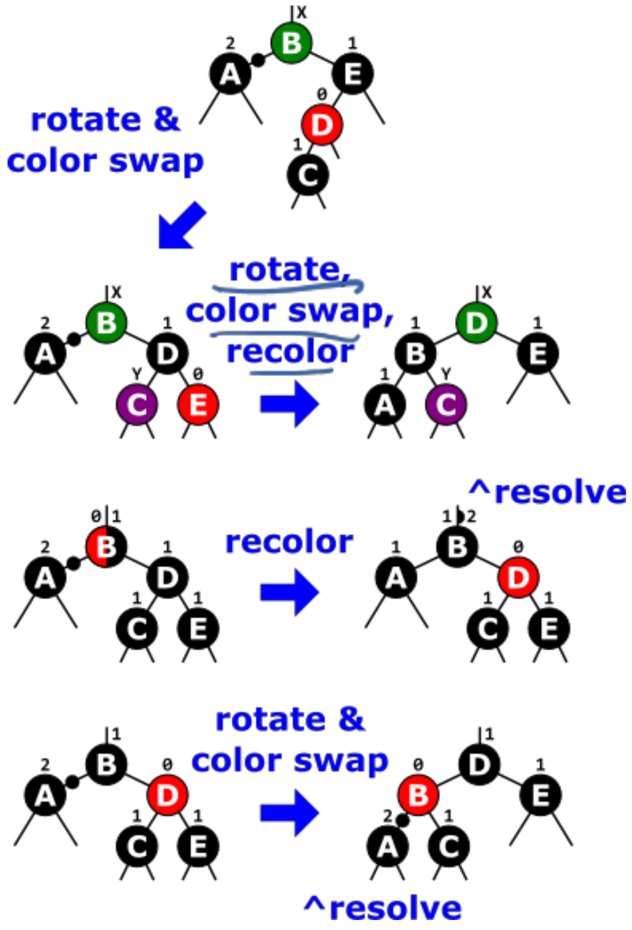


1) left rot 7,9

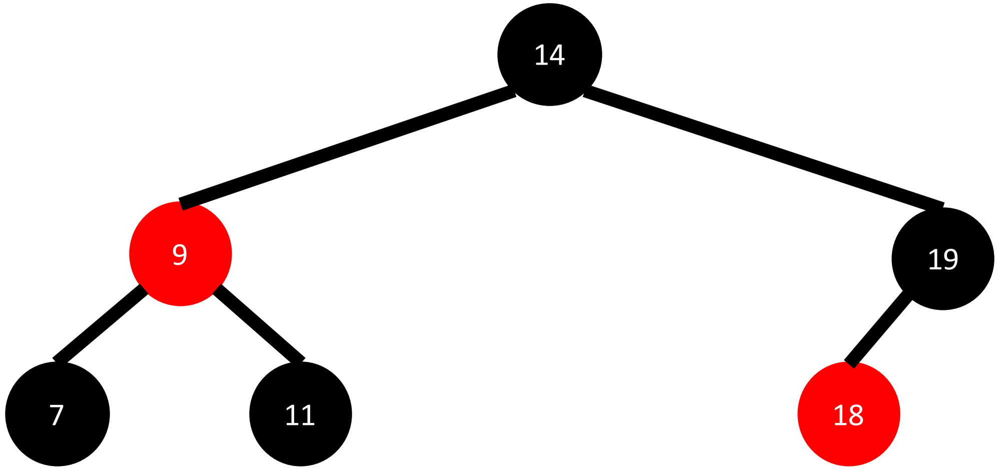


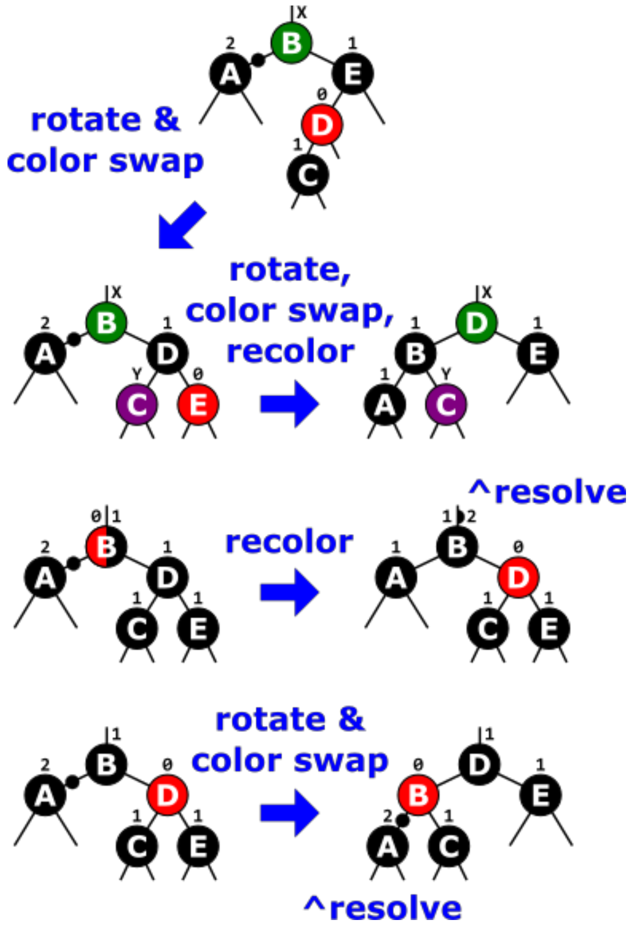
2) color swap 7,9 + recolor 11



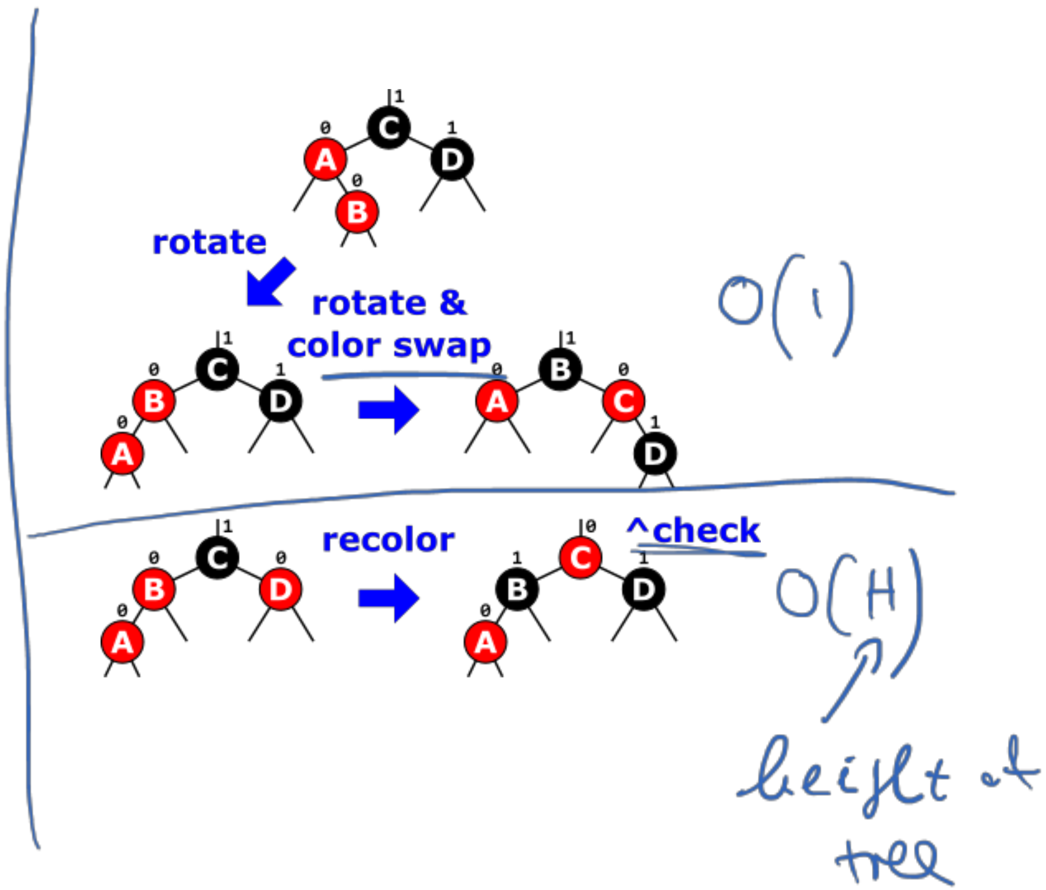


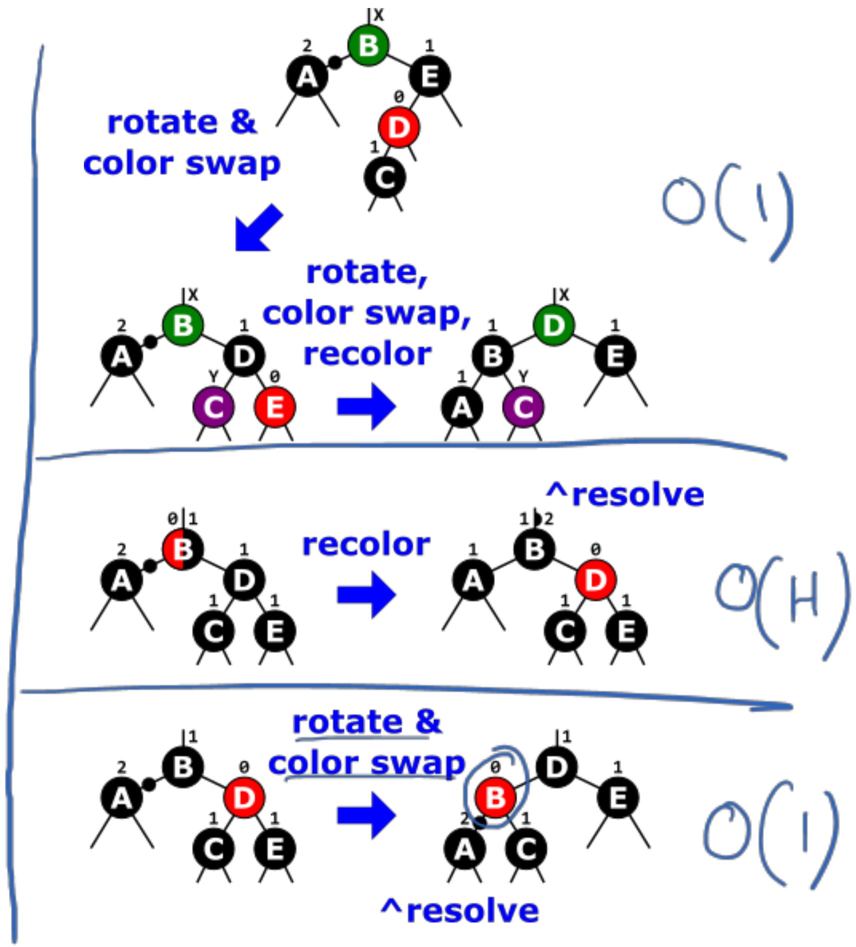
BLACK Node Without RED Replacement





Complexities of Repair Operations





Complexities of RBT Search, Insert, and Delete

H: height of tree
H_b: black height of tree
N: number of nodes in tree

$$N \geq 2^{H_b} - 1 \quad \Big| \quad H \leq 2 \cdot H_b$$

$$\log_2(N+1) \geq H_b \quad \Big| \quad \frac{H}{2} \leq H_b$$

Search: $O(H)$

$$O(\cancel{2} \cdot \log_{\cancel{2}}(N \cancel{+1})) \stackrel{?}{=} O(\log N) \qquad \log_2(N+1) \geq \frac{H}{2}$$

Insertion: $O(H \cancel{+H}) \stackrel{?}{=} O(\log N) \qquad 2 \cdot \log_2(N+1) \geq H$

Delete: $O(H \cancel{+H}) \stackrel{?}{=} O(\log N)$